

Design and Maintenance Specification

FOR

CTG

Reachability & Control Subsystems

Revision 1.0

Date:02/14/00

AUTHOR(S):, Kevin Harer

REF ID: A5003420

1.	Introduction	3
1.1.	Document Scope.....	3
1.2.	Intended Audience	3
1.3.	Document Organization.....	3
2.	CTG Usage Overview	3
3.	Architectural Overview.....	5
3.1.	Master/Slave Infrastructure	5
3.2.	TCL Control Mechanisms.....	6
3.3.	UI Interfacing	6
3.4.	Reachability Engines.....	6
3.5.	Unreachability Engines.....	7
3.6.	CTG Process Flow	7
4.	Master Process Coordination Mechanisms	9
4.1.	Process Hierarchy and Communication.....	10
4.2.	Infrastructure Package (hvCtl)	10
4.3.	TCL Coordination Package	14
5.	Goal Creation and Usage.....	35
5.1.	Introduction.....	35
5.2.	HvGol Package TCL Commands	35
5.3.	HvPli Services For VCS Usage	36
5.4.	Usage Scenarios	38
6.	UI Interface.....	39
7.	VCS Interface	39
7.1.	PLI Interfacing – hvPli	39
7.2.	Verilog TestBench Top Module.....	41
7.3.	Vera Interfacing - CoverBooster.....	44
7.4.	Verilog Program Control – Usage Scenarios	48

1. Introduction

1.1. Document Scope

This document serves as a design and maintenance document for the Reachability & Control Subsystem of the "Coverage-based Test Generation" product (CTG). This document describes various mechanisms, techniques, and policies which are used to coordinate formal and informal engines as a solution to our specific reachability problem. This includes interfaces to the engines and integration techniques for those engines.

This document does not describe in detail individual engines or underlying infrastructure of the formal analysis environment. The document does not describe detailed User Interface issues, mechanisms, methodologies, or flows. VERA environmental modeling and compilation is not discussed. Refer to appropriate documentation for those other technologies.

1.2. Intended Audience

The intended audience is technical engineering staff which are actively involved in CTG development. The document may be useful for non-technical persons who are familiar with CTG problems and issues.

1.3. Document Organization

The remainder of this document is organized in the following way. Section 2 provides an overview of the problem CTG addresses and the general approach from a users perspective. Section 3 provides an architectural overview of the CTG system implementation. Section 4 provides detailed descriptions of mechanisms used to coordinate, control, and/or integrates the various engines and UI into a cohesive system. Section 5 provides detailed descriptions of goal creation and maintenance in CTG. Section 6 provides detailed interfacing mechanisms for the UI process. Section 7 provides detailed descriptions of the interface with the explicit states simulator VCS, as well as detailed explanation of test generation and replay mechanisms and capabilities.

2. CTG Usage Overview

The basic problem that CTG attacks is the following. Given some Design Under Test (DUT), and some goal states of that design, classify all goals as either Unreachable or Reached. If a goal is classified Unreachable, it must be proven with respect to some start state and DUT environmental conditions. If a goal is Reached, then CTG must save a simulation test suitable to reach that goal in some later VCS stand-alone session. Any goal which is not classified as Reached or Unreachable is said to be Unknown; CTG attempts to minimize the number of these Unknown goals.

CTG uses a state coverage metric, hence goals are states, or sets of states, of the design. These states are defined by a small number of "coverage variables". These coverage variables are on the order of dozens, while the total latches in the design are in the thousands. Each goal is a different combination of these coverage variables. As an example; given coverage variables {c1, c2}, CTG attempts to reach "states" {00}, {01}, {10}, {11}. Note that these "states" are technically "cubes", but we call them states.

The CTG objective then is to classify as many different combinations of these coverage variables as possible. This task is known as "Coverage Driven Test Generation".

The method CTG uses to "reach" (or cover) goals is to interleave classic explicit state simulation (e.g. VCS) with formal exhaustive searches using some explicit design states as starting points for the search. If a formal search is successful, then the explicit state simulator is used to build up a simulatable test which reaches the goal. This scenario is illustrated in Figure 1. In Figure 1, the enclosing box represents the entire state-space of the design. Explicit state simulation is represented by the zig-zag solid line as explicit

states are visited on each successive clock cycle. Stars represent goals which are to be “reached”, if possible.

At some time T1, CTG determines that simulation is not reaching new goals. A formal search is used to exhaustively search for a goal from some explicit state, S1, generated by the simulator. This formal search is represented by the concentric rings in Figure 1. When a formal search reaches a goal G1, the sequence of input assignments (termed a “trace”) to reach from S1 to G1 are simulated and the goal is then “reached”.

This interleaved process continues until all goals are classified or some user-specified resource limit is exceeded (typically time).

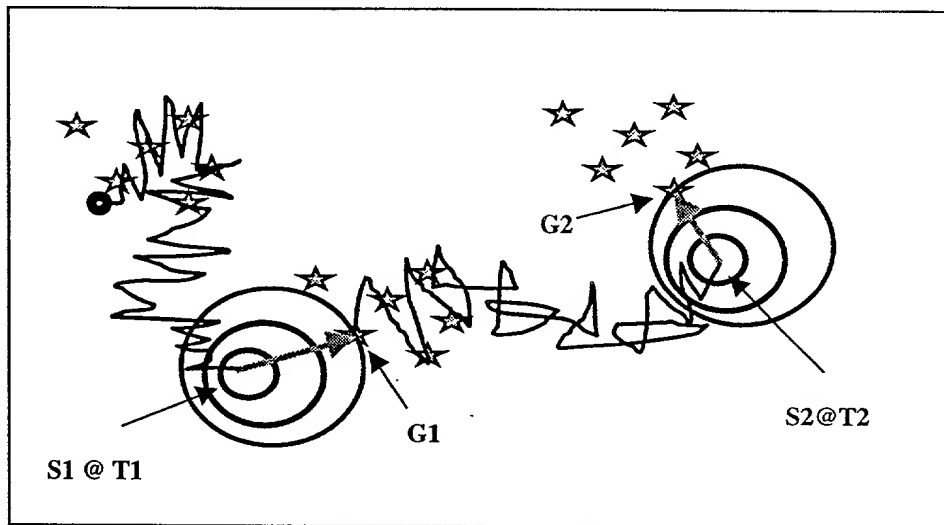


Figure 1. CTG Search Process

Some basic observations which lead to this interleaved engine usage are the following:

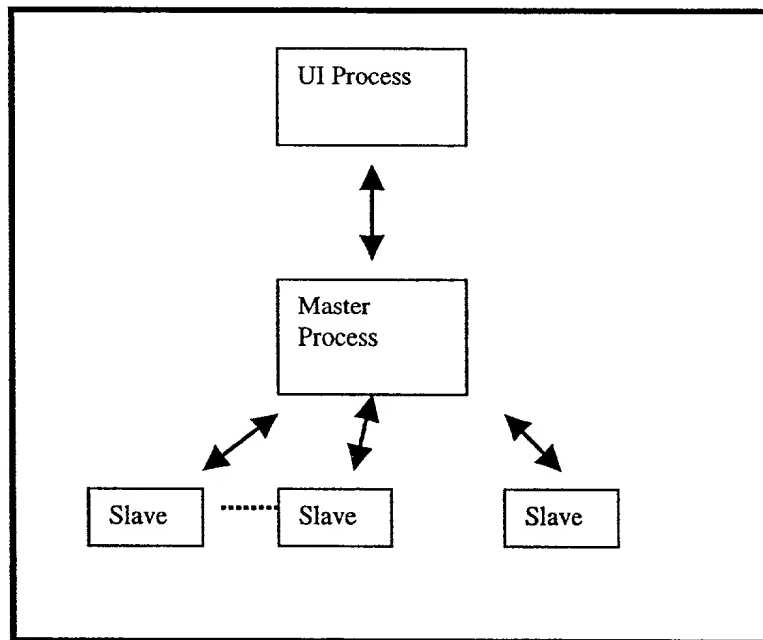
- Explicit state simulators are very efficient at reaching “deep” states in the design. Explicit state simulation only reaches a single state at each clock cycle.
- Formal search methods are exhaustive, which means that all reachable simulation states are reached at the same time. Formal reachability algorithms are feasible for short search radius, and commonly intractable for deep search radius.
- When a difficult goal is reached, there are often several other goals “close” to that goal in the state-space. This is termed “jackpot” behavior. The rationale for this is due to the nature of the goals that CTG attempts to reach. As described earlier, goals consist of a collection of coverage variables. These coverage variables are the union of smaller control Finite State Machines (FSM’s) in the design. Some of these FSM’s are relatively easy to transition to new states, others are relatively difficult. When a difficult FSM state is entered, the cross states determined by the easy FSM’s are then easily reached; these are the jackpot goals.

3. Architectural Overview

This section provides a general description of large components of the CTG system which accomplish the usage described in the previous section.

3.1. Master/Slave Infrastructure

The overall CTG architecture may best be thought of as a Master/Slave where a Master process interacts and coordinates many concurrent Slave processes. These Slaves are used to “reach” various goals, prove them unreachable, and to interact with the user. Fig.



2 illustrates this architecture.

Figure 2. CTG Process Architecture

As illustrated in Figure 2, a “Master” process is used to coordinate more than 1 Slave (or child) processes so that the User Interface process is isolated from all control/coordination problems. The Master process is also given the task of providing communication mechanisms between the Slaves and UI process. In addition, various system heuristics are embodied in the Master process to choose which Slave engine to use at different times in the CTG session.

It is important to note that the Master process is intended to be “live” at all times; the Master process does not “block” for long times while various analysis are carried out by any given Slave. An additional requirement is that the Master process allow Slaves to work concurrently.

The Master process continually polls the Slave and the UI processes for messages and/or commands to be processed. When a message is received from a Slave, it is decoded and either processed by the Master process itself, or passed to the UI. Commands from the UI are processed immediately by the Master process.

The Slave processes of Figure 2 represent various reachability and/or unreachability engines, each running in a separate unix process. Examples of these engines in the CTG system include: symbolic simulation, discrete event simulation (VCS), satisfiability solver, and an image computation engine. Depending on usage, it is possible for 2 or more of these engines to be used in parallel within a single CTG analysis session.

3.2. TCL Control Mechanisms

Control flow for the CTG product is embodied in TCL procedures which execute in the Master process. This TCL control code interacts with Slave processes and the UI process to coordinate all Slaves into a coherent CTG session.

Each Slave process receives commands from the Master process via stdin of the Slave process. The Slave communicates with the Master by writing properly formatted text strings to stdout, or by putting large amounts of data (e.g. BDDs) in data files formatted for the specific task at hand.

Two classes of messages are received by the Master process from Slave processes: formatted and unformatted. Unformatted messages are simply printed to stdout of the Master process. Formatted messages from the Slave are, by definition, intended to be processed by the TCL control program running in Master. The Master process transfers this information from the unix/C world to the TCL world by calling a TCL procedure; this procedure then further decodes the message for handling by the TCL control program. A message "catalog" is maintained such that all pre-defined (thereby formatted) Slave messages are known.

3.3. UI Interfacing

The UI process issues commands to stdin of the Master process, and receives formatted messages from stdout of the Master process. The UI process is responsible for all user presentation of data received from the Master process. This presentation may be either graphic or textual in nature, depending on user needs. A UI message catalog is maintained.

3.4. Reachability Engines

CTG uses a number of reachability engines, each running in its own unix Slave process. Each of these engines use specialized algorithms to search for sequences of inputs which lead from one state (a "start" state) to others ("goal" states). Examples of these reachability engines include:

- Explicit state engine. This is a classic event driven or cycle-based simulator. This engine uses biased, constrained, random simulation techniques to exercise the Design Under Test (DUT) in an attempt to reach goal states. In addition, this engine has the capability to:
 - generate start states for searches performed by other, more "formal", engines,
 - apply sequences of inputs which are generated by the formal engines,
 - save the entire testcase for later re-use by the user
- Symbolic Simulation engine. This engine uses exhaustive (perhaps under-approximated) symbolic simulation methods to attempt to reach goals from specific starting states. If a goal is reached, the engine saves a sequence of

inputs which are replayed by the explicit state engine to prove that the state has been reached.

- Satisfiability Solver engine. The SAT engine uses bounded model checking concepts, together with classic satisfiability techniques, to reach goals from a starting state. If successful, the engine saves a sequence of inputs which are replayed by the explicit state engine to prove that the state has been reached.

3.5. Unreachability Engines

Unreachability engines use various formal techniques to prove that certain goal states are not reachable from some start state. The start state of interest is typically the reset state of the DUT. These Unreachable states are then optionally communicated to the reachability engines to reduce the search problem. Current unreachability engines include:

- Image Computation Engine. This engine computes an approximate fixed point for reachable states, then inverts the fixed point as a proof of unreachable states. The fixed point computation is typically over-approximated.

3.6. CTG Process Flow

An illustration of the interleaved and concurrent nature of the CTG system is illustrated in Figure 3.

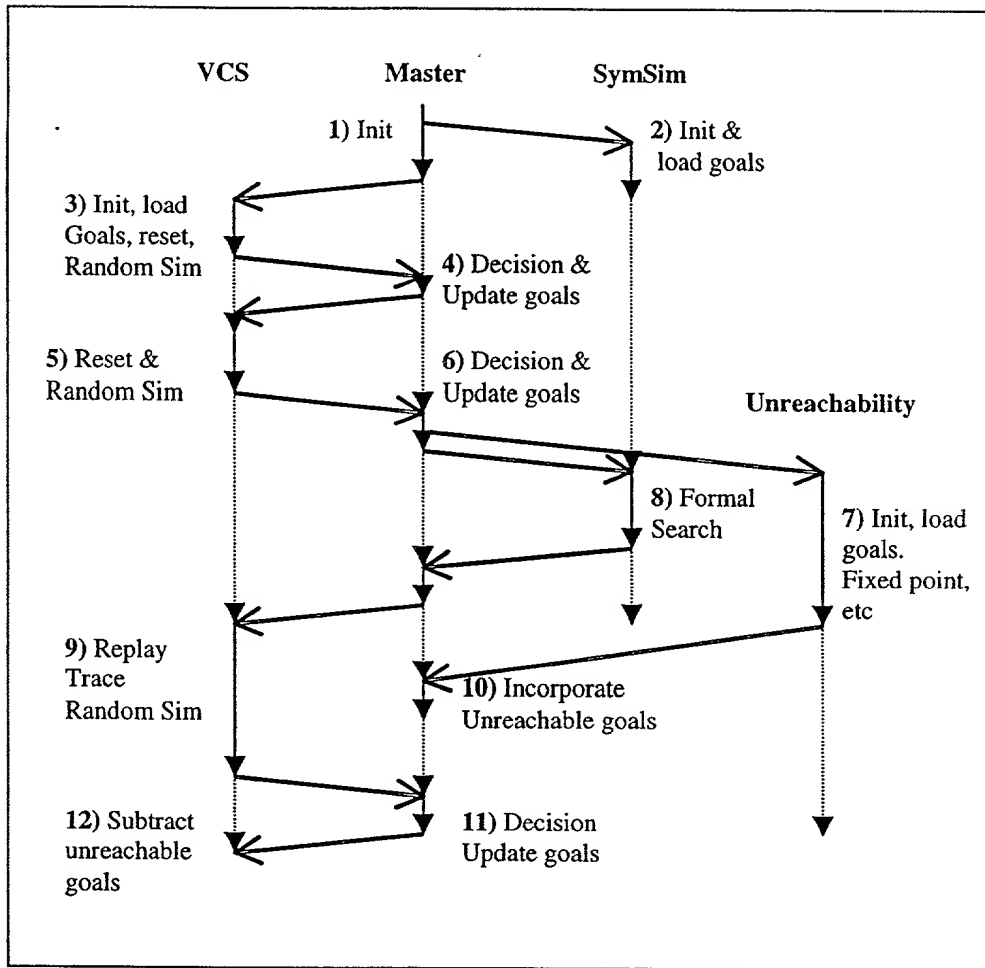


Figure 3. Interleaved CTG System Overview

The diagram of Figure 3 shows an abbreviated, simplified CTG session where VCS, Symbolic Simulation, and Unreachability analysis collaborate to classify goals. The vertical solid arrows indicate that the unix process is "working". The vertical dashed arrows represent that the unix process is waiting for input from some other process before continuing. The horizontal double-arrows indicate messages and/or data being passed between unix processes.

A description of some of the interesting transactions and tasks follows. Note that all UI interactions are omitted, as they are specific to another document.

1. The first thing that happens is that the Master process is invoked. After invocation, the Master process loads HNL, defines goals, and generally does initialization type things.
2. At some point, based on input from the user, the Master process starts and initializes formal reachability engines in separate unix Slave processes. In this example, only the Symbolic Simulation engine is used. The hv program which embodies this reachability engine loads HNL, defines goals, and builds the Symbolic Simulation Manager.
3. The Master process, again under UI direction, starts the VCS Slave process. VCS initializes, loads goals, performs a reset sequence, then runs a number

of random simulation cycles. At this point, the CTG-resident code in VCS is keeping track of which goals are being "reached". At some point, CTG PLI code in VCS determines that "goals are not being reached". VCS stops and returns control to the Master process.

4. The Master process, based on various heuristics and user parameters, makes a decision on how to proceed. In this example, the decision is to perform a reset sequence then do more random simulation in VCS.
5. VCS performs a reset sequence to reset the DUT, then more random simulation occurs. After determining that no more goals are being reached, VCS stops simulating and returns control to the Master process.
6. The Master process decides to perform a formal search, using Symbolic Simulation, from the current design state. This design state is written from VCS and is used to 7) start unreachability analysis, and 8) start a formal search.
7. An Unreachability Slave is started. This is performed by the hv program, which loads HNL and goals, then performs a fixed point computation from the design state provided from VCS. After the fixed point is computed, those goals outside the fixed point are proven unreachable. These unreachable goals are written to disk for later processing by other Slave engines.
8. Simultaneously with unreachability analysis, a formal search is initiated using the Symbolic Simulation engine. This search starts from the design state captured by VCS. In this scenario, the search completes successfully, and the Symbolic Simulator writes out primary input stimulus (i.e. a "trace") which VCS can use to drive the design from the current state to the goal state.
9. VCS replays the trace and hits the new goals. Random simulation continues in an attempt to hit the previously described "jackpot" goals.
10. While VCS is performing random simulation (trace replay, etc), the unreachability engine completes and results are incorporated into the Master process. This unreachable goal data will be communicated to each of the reachability engines on an as-needed basis.
11. At some point, VCS determines that no more goals are being hit, and control is returned to the Master process.
12. The VCS process is sent a message which contains the number of unreachable goals. This count is used to adjust the coverage metric which tells VCS when all goals have been classified.

The CTG session continues interleaving VCS and Symbolic Simulation until all goals are reached. The remainder of this document explains the details about how these things are implemented.

4. Master Process Coordination Mechanisms

As illustrated in Fig. 2, the Master process is the central coordinator of a multi-process concurrent system which attempts to classify goals in some sort of efficient manner. TCL procedures are implemented to

provide key coordination of the Slave engines, and to embody heuristics to attempt to provide an efficient coherent system to the UI. Important infrastructure services are provided in C code packages to provide flexibility to the TCL control programs. The hv program provides this Master process by invoking it the following way:

- `hv -I -master`
- Typically this invocation is performed by the UI process which 'fork's a unix process, then 'exec's the hv executable in it.

This section presents detailed descriptions of the mechanisms which accomplish the Master process capabilities.

4.1. Process Hierarchy and Communication

The process hierarchy of Figure 2 is the following. The Master process is started by the UI process. Slave processes are started by the Master process. These processes are started using standard unix 'fork' and 'exec' mechanisms.

When a child process is started, the parent process redirects standard-in and standard-out of the child process into unix pipes which are then read and written by the parent process. Thus communication from the parent to the child is accomplished as the child process reads from its stdin file descriptor. The Master and most Slave processes execute the hv program which processes stdin using the Synopsys cci TCL interpreter. The VCS Slave uses the VCS interactive interpreter to read from stdin. Thus all messages to child process must be as either TCL hv commands or VCS commands, depending on the child.

Messages from a child to a parent are communicated via writing to stdout of the child process. The parent process must periodically read from the earlier-mentioned pipe to retrieve and process these messages from the child.

In some cases, where large amounts of data are needed to be communicated between processes, it has proven useful for the data provider to write the data to a temporary file. The consumer is then sent a message saying what/where the data is. The consumer of the data opens the temporary file and processes the data.

4.2. Infrastructure Package (hvCtl)

The hvCtl package provides the key mechanisms to start Slave process, coordinate them, and relay messages to either the UI process or TCL procedures resident in the Master process. Services are provided to allow the TCL programs to communicate and track progress of the Slave processes. These services are accessed exclusively by TCL code which runs in the Master process. This section presents key concepts and commands which access the implementation of those concepts.

4.2.1. Channels

A "Channel" defines a communication path between the Master process and a Slave process. Each Channel has a unique ID which is assigned at the time the Slave process is started. This unique ID is then used to send commands to that Slave and to identify messages received from the Slave. Commands to manipulate and interact with a Channel are the following:

- `hvctl_channel_open` – This command opens a channel by starting a user-specified program in a separate unix process. The Master process then sends messages to the Slave via the `hvctl_send_message` command.

Messages/responses from the Slave will then be read from stdout of the child process. Upon success, this command assigns a unique ID for for TCL usage to identify the new Channel. Input arguments to the command allow specification of the program to start in the new process, the working directory the process is to run in, and up to 6 arguments to be used in invocation of the program which is started.

- **hvctl_channel_close** – This command closes a channel and cleans up data structures associated with it in the Master process. Input arguments are the Channel ID, and an optional command which terminates the program running in the Slave process.
- **hvctl_channel_send_msg** – This command sends a message to the program running in the Slave process. Input arguments are the Channel ID and a string which is sent to stdin of the child process. The message string must be a syntactically and semantically correct command for the program running in the Slave process. The message is presented to that program on stdin of the Slave process.
- **hvctl_channel_stack_set** – Due to the concurrent, non-blocking nature of the CTG system, a mechanism is needed to allow long-duration Slave commands to execute. The problem for the controlling TCL program then becomes “remembering” what is happening in a Slave process over a long period of time, and what needs to be done when that task is finished. The mechanism provided is termed a “Command Stack”, which is described later in this document. This command associates a Command Stack with a specific Channel. Input arguments are the Channel ID and the name of the Command Stack to associate with it.
- **hvctl_channel_stack_get** – This command returns the name of a Command Stack associated with a specific Channel. If no Command Stack has been assigned, the string “<no_stack>” is returned.
- **hvctl_channel_dump_info** – This command is provided for debugging purposes and prints various information about an open Channel.

4.2.2. Engines

An Engine is a “bookkeeping” aid which exists in the Master process, and defines a mechanism which is resident in a Slave. For example, a “Symbolic Simulation Engine” object might exist in the Master process, and correspond to an entire Slave process which is used to perform Symbolic Simulation.

This corresponding Slave mechanism performs some sort of analysis or task which may or may not involve a number of individual commands/responses between Master and Slave. It is possible (and common) to have multiple Engines associated with a single Slave process at the same time. Examples of Engines are the Symbolic Simulator or SAT solver.

Note that the Engine concept is very general, and it has proven useful to have more abstract Engines such as the Goal Manager, the Design-Under-Test Manager, etc.

Engines may have state; for example, the Symbolic Simulation Engine might be in the “RUNNING” state, or the “IDLE” state. This state concept has proven useful to help the controlling TCL program deal with the non-blocking, concurrent nature of CTG.

- **hvctl_engine_type_define** – This command defines a new *type* of Engine. Typically all Engine types are defined at Master process startup.
- **hvctl_engine_state_define** – This command defines a legal state for the newly-defined Engine type. Valid Engine states are typically defined at Master process startup, when the new Engine type is also defined.
- **hvctl_engine_start** – This command creates an *instance* of an Engine type, associated with a specific Channel. The user specifies Engine type, Channel ID, and a name the Engine instance will be known by. The combination of type, Channel, name must be unique.
- **hvctl_engine_state_set** – This command assigns a state to an instance of an Engine. The state must be valid (defined with `hvctl_engine_state_define`) for the Engine type. If the specified state is invalid, an error occurs. The caller must specify a unique combination of Engine type, name, and Channel ID.
- **hvctl_engine_state_get** – This command returns the current state of a specific Engine. The caller must specify a unique combination of Engine type, name, and Channel ID.
- **hvctl_engine_exists** – This command returns a space-delimited list of names of current Engines which match caller specified parameters. For example, the caller can request “All Engines defined for Channel ID=4”, or “All Engines of type Symbolic Simulator”. If no Engines are found, the empty string “” is returned.
- **hvctl_engine_stack_set** – This command is analogous to the previous `hvctl_channel_stack_set` command. The caller specifies a Command Stack, and unique combination of Engine type, name, and Channel ID.
- **hvctl_engine_stack_get** – This command returns the name of the Command Stack associated with a specific Engine. The caller must specify a unique combination of Engine type, name, and Channel ID.
- **hvctl_engine_type_dump** – This is a debugging command which prints various information about a valid Engine type.
- **hvctl_engine_dump_info** – This is a debugging command which prints information about a current Engine instance.

4.2.3. Command Stacks

As mentioned earlier, the non-blocking nature of the Master process leads to an interesting problem when the Master asks the Slave process to execute a time-consuming task. The TCL control program cannot launch a long task, then simply block-and-wait for the task to finish before continuing to the next task in a sequence of tasks which need to be completed. The TCL program must not block; this non-blocking requirement allows for interactions with the UI or other Slaves. Yet somehow, the TCL program needs a convenient mechanism to help remember what to do when the time-consuming Slave task finishes. The Command Stack is the key mechanism provided to assist in this problem.

A Command Stack is a stack of commands which may be attached to either a Channel or an Engine. When a task is finished (detected via a message from the Slave), the next command is popped from the stack and is executed. Common usage by the TCL control program include the following:

- A new Command Stack is created and associated with each Channel when the Channel is opened.
- When a time-consuming task is started by a Slave, a TCL procedure, with necessary arguments, is pushed onto the Command Stack for that Channel. This pushed TCL procedure is to be executed when the time-consuming task is completed.
- Optionally, the state is set on the Engine which will signal completion of the time-consuming task.
- When the task-is-done message is received by the Master process, the command is popped from the Command Stack, and it is executed. Note that this Slave-to-Master message passing is explained later in this document.

The remainder of this section describes commands which directly manipulate Command Stacks.

- **hvctl_command_stack_create** – This command creates a new Command Stack. The caller may optionally assign a name to the Command Stack – if so, the name must be unique.
- **hvctl_command_stack_free** – This command destroys a previously-created Command Stack.
- **hvctl_command_stack_push** – This command pushes a syntactically legal TCL command string onto the command stack. This string is executed at some later point using the TCL “eval” mechanism.
- **hvctl_command_stack_pop** – This command pops the top command string off the Command Stack. If the stack is currently empty, the string “<empty>” is returned.
- **hvctl_command_stack_dump** – This debugging command prints information about a current Command Stack.

4.2.4. Message Poll Loop

As illustrated Figure 2, the Master process needs to continually check for messages from the UI process as well as each of the Slave process. This mechanism uses the unix “select” mechanism to accomplish this. “select” is a unix function which is given a set of open file descriptors to attempt to simultaneously read from. When data is read from one of the files, select returns with the file descriptor of the file with waiting data. The select function has the ability to also “time out” if no file may be read after a certain amount of time.

For the Master process, these files are either stdin of the Master process, or the open “pipe” (another unix-ism) for individual Slave processes. These pipes are created when a Slave process is started, and correspond to stdin and stdout of the Slave process.

When data is read from one of these files, the Master process first determines which file the data is read from. If the file is the output of a Slave process, the text is decoded to determine if the message is formatted in such a way as to be intended for the TCL mechanisms in the Master process. Formatted Slave messages are identified by prepending "HVINFO" and a message ID to the actual data. The message ID is associated with a specific engine and is expected to be understood by the TCL control program in the Master process.

Once the active file and message decoding has been performed, the Master process takes the following action:

- If the data is read from stdin, it is a TCL command from the UI process which needs to be executed. The Synopsys cci command interpreter is used to execute these commands.
- If the data is read from a Slave process, the message is either unformatted or formatted. Unformatted text is relayed to the UI by simply writing the text to stdout of the Master process. Formatted messages are communicated to the TCL control program in Master by calling a predefined TCL procedure. The TCL procedure is called "hv_child_msg_rcv" and arguments include
 - Channel ID for the Channel which sent the message, the message ID, and the actual text of the message.

4.3. TCL Coordination Package

The CTG control policies, mechanisms, and heuristics are all implemented in TCL procedures which are executed by the "hv" program running in the Master process. This TCL control code must issue commands to the Slave programs, interact with the UI process, track progress of the test-generation session, etc. This task is complicated, and key aspects are described in this section; it is beyond the scope of this document to explain all necessary details.

Note also that this TCL code is dynamic in nature (i.e. it changes often) as heuristics and algorithms change.

4.3.1. Messages and Message Handlers

As previously discussed, when the Master process reads a message from a Slave process, this message may need to be communicated to the TCL control program. This is done by a call to the TCL procedure "hv_child_msg_rcv" which includes a unique message ID, the Channel ID the message is from, and the actual message itself.

Messages are defined on an Engine-specific basis, and are used to communicate some data and/or status from a Slave to the TCL control program in the Master process. These messages must obviously be unique, and coordination of message ID/content between the Slave implementation and Master TCL code must be coordinated by the developer of the engine (Slaves issue messages, Master receives & processes them). To help with this coordination, each Engine has a separate TCL procedure where all messages for that Engine are defined and handled. The hv_child_msg_rcv procedure decodes message ID enough to decide which Engine-specific message handler to call, then that procedure is called with the message for further processing.

Message Ids are textual, and the collisions are avoided between engines by assigning a unique prefix to all messages for that engine. For example, all messages from the VCS engine are of the form "VCS_<MsgName>" where <MsgName> varies for each message

from the VCS engine. These textual prefixes are documented for all known engines in the hvRecipe.tcl file.

(Developers note: At time of writing this document, message Ids are numeric, and the actual message format includes a data type field. This will soon be modified to reflect the format described in this document.)

To add a new Engine, the developer must perform the following tasks:

1. Create a file <Engine>Handler.tcl and put it in /vobs/propver/src/tcl. This file will implement the Engine-specific message handler TCL procedure hv_<Engine>_child_msg_rcv. This file also contains the Engine type and valid states definitions. Allocate a textual message ID prefix for the new Engine, and document it in hvRecipe.tcl.
2. Modify the hv_child_msg_rcv procedure in hvRecipe.tcl to recognize your new messages and call your new procedure when a message in the proper range is detected. Source the <Engine>Handler.tcl file where all the other message handlers are sourced, at the end of hvRecipe.tcl.
3. Document your new messages and handle them by implementing the hv_<Engine>_child_msg_rcv procedure in the <Engine>Handler.tcl file. Stylistically, use one of the pre-existing *Handler.tcl files in the tcl directory as a template. By convention, this file does not "know" about other engines or high-level procedures. This message handler typically interprets messages, sends notice to the UI, sets engine state, then pops high-level commands off the command stack to continue the CTG session.
4. Instrument the corresponding Slave implementation of the Engine to generate messages in a semantically appropriate manner.
5. Modify the control program TCL code in hvRecipe.tcl to use the new Engine in an appropriate way. This code is discussed in the following section.

At the time of writing of this document, a certain number of Engines have been defined and integrated. Those engines and their usage are:

- Symbolic Simulation Engine. This Engine is implemented in the hv executable and performs Symbolic Simulation from some start state in an attempt to reach some pre-defined goals. If a goal is reached, a "trace" is written for VCS use which applies a sequence of input to drive the design from the start state to the goal state.
- VCS Simulation Engine. This Engine is implemented via PLI code attached to the VCS simulator. This Engine is responsible for observing which goals have been reached as simulation progresses, detecting points at which random simulation should stop, testcase generation and replay, and interfacing with the user-written environmental biasing and constraint code. In addition, this Engine is capable of conveying search start states to the formal reachability engines.
- SAT Engine. This Engine is similar to the Symbolic Simulation Engine except that satisfiability techniques are used to attempt to find a sequence of

inputs from some start state to a goal state. This Engine is implemented in the hv executable.

- Image Computation Engine. This Engine uses symbolic state traversal techniques to prove that certain states are unreachable from specific start states. These provably unreachable states are then used to classify goals which are unreachable. This Engine is implemented in the hv executable.
- Goal Engine. It has proven useful to define a Goal Engine to manage goal creation, update, and coordination between the various Slave process and the Master process . This Engine is implemented in the hv executable.
- DUT Engine. A DUT Engine is used to manage the design load and pre-processing steps which lead up to reachability and/or unreachability analysis. This Engine is implemented in the hv executable.

4.3.2. Session Startup

Refer to a UI-related document for detailed description of user interactions with the CTG system (i.e. “how the User interacts with CTG”). In general, the user will first setup an analysis session by doing things like creating hnl, compiling the simulation model, defining goals, environmental modeling, etc. Following this, the user clicks on “Generate Testcase” and the following sequence of events take place:

- The UI starts the Master process by forking a child process and invoking “hv -I -master” in that process. The UI opens a pipe to read/write stdout/stdin of the Master process.
- The UI issues hv commands to setup the Master process. These commands must accomplish the following:
 - The hvRecipe.tcl file needs to be sourced
 - User-specified session parameters (i.e. TCL knobs) must be set. These parameters are discussed in a later section, but this step basically sets a number of pre-defined TCL variables.
 - The HNL must be loaded into the Master process by calling the hvino_create_hnl_design command.
 - The HvDsg must be created, and any necessary preprocessing must be accomplished. This is done using the hvdsg_create_hv_design command. Note that current HvDsg usage is that both a hierarchical and flat HvDsg are created in the Master process. All formal and analysis is performed using the flat design; the hierarchical design is used for interaction with the user. Commands are provided for mapping between flat and hierarchical worlds.
 - The proper HvMch should be created using hvnmch_create_machine.
 - An appropriate cover (HvCic) should be created using hvnmch_create_cover

- An SthMgr and OrdMgr must be created using hvsth_create_manager and hvord_create_manager
- The user-specified goals are created in the Master process.
- The UI process requests a Slave process to be started for SAT and/or Symbolic Simulation analysis. This is accomplished by a call to the “hv_setup_formal” TCL procedure. This procedure is defined in hvRecipe.tcl and described later in this document. A Slave process is started, hnl loaded, goals defined, and any algorithm-specific setup is performed (e.g. create the Symbolic Simulation Manager).
- The UI initiates actual test generation by launching VCS in its own Slave process. This is done by calling the TCL procedure: “hv_start_vcs_session”. This procedure is defined in hvRecipe.tcl and described later in this document. A Slave process is started and the compiled simulation model (generated by VCS compilation earlier) is started. Goals are loaded in VCS and a reset is performed, then simulation pauses. The Master process then sets up simulation stop conditions (via calls to the CTG PLI code) and initiates a simulation run from the reset state.
- At this point, the CTG PLI code within the simulator is recording goals which are reached, as well as any information needed to later write out the final testcase.

From this point on, the CTG session progresses according to the “Phase Progression” described in the following section.

4.3.3. CTG Analysis “Phase” Progression

After the reachability session is initiated, the resultant actions are highly dependent on various heuristics, user-specified direction, and how “hard” the design and/or goalset is. In general, the session progresses through a number of phases or modes. These phases are designed to in general be progressive in CPU and Memory cost.

This section describes the phase progression. Search Engine usage in those phases are described as are exit conditions for the phase. Note that this progression is the heart of CTG and can be expected to change “A LOT” as CTG is exposed to more designs and the development team learns how to effectively use all the technology. This section is provided to give the reader a feel for how things worked at least once in the past.

A CTG reachability session transitions through the following phases in order. At any point, if all goals have been reached, then the testcase is written and the system stops. In the following discussion, certain user settings (i.e. “Knobs”) are referenced; these knobs are cataloged and described elsewhere in this document. In the following discussion, knobs are italicized and in bold.

- **MODE_INIT**

This is the initial mode that CTG starts in. The VCS Engine is used to repeatedly reset the design followed by a number of biased random simulation cycles. The number of times to repeat this sequence is specified by *initRunsMax* and the number of simulation cycles to run is specified by *initCycles*. *initCycles* specifies that VCS will stop when this many system clock cycles are seen without hitting a previously unreachable goal.

When this initial run-from-reset phase is finished, VCS performs a final reset then stops. The reset state is written to a file by VCS. If requested by the user by setting the *unreach* knob to "UNREACH_LMBM", an unreachability analysis is initiated.

If *satCycMax* is > 0 , CTG progresses to MODE_INIT_SAT, else MODE_INIT_SYM.

- MODE_INIT_SAT

In this phase, CTG uses the SAT engine to search for goals from the reset state. SAT uses the resource limits *satCycMax* and *satCpuMax* to know how hard to search.

If a goal is reached, the trace is replayed in VCS followed by random simulation until *initCycles* system clock cycles are seen without hitting a new goal. Then VCS resets and waits for another formal search to be carried out.

If SAT fails to reach a goal from the reset state, then CTG progresses to the next phase. If *symsimCycMax* > 0 , then MODE_INIT_SYM is entered. Otherwise CTG enters MODE_SAT.

- MODE_INIT_SYM

This phase is entirely the same as MODE_INIT_SAT, except the symbolic simulation search engine is used. Resource limits are defined by *symsimCpuMax* and *symsimCycMax* knobs. Upon a formal search miss, CTG switches to the next phase. If *satCycMax* > 0 , then MODE_SAT is entered; otherwise MODE_SYMSIM is entered.

- MODE_SAT

In this mode VCS simulates to a deep state (as illustrated in Figure 1) and stops. This deep state is written to file and is used for a formal search using the SAT engine. SAT uses resource limits *satCycMax* and *satCpuMax*. If SAT reaches a goal, the trace is replayed in VCS followed by more random simulation.

In each case, VCS stops to try a formal search after *rsimCycles* are seen without hitting a unreachable goal, or when a "fresh" state is seen. The definition and detection of fresh states are described later in this document.

After *formalMissThresh* consecutive search misses, VCS is reset. If *symsimCycMax* > 0 , then MODE_SYMSIM_SAT is entered, else CTG stays in this phase forever.

- MODE_SYMSIM_SAT

This phase is similar to MODE_SAT, in that formal searches are performed from deep states generated by VCS. VCS stops in the same manner as that described in MODE_SAT. The difference is that SAT and Symbolic Simulation searches are alternated. When *formalMissThresh* consecutive formal misses are seen, the design is reset and CTG enters MODE_SYMSIM.

Resource limits for the formal search engines are *satCycMax*, *satCpuMax*, *symsimCpuMax*, and *symsimCycMax*.

- MODE_SYMSIM

This phase is entirely like MODE_SAT, except that Symbolic Simulation is used instead of SAT.

- `MODE_DONE`

CTG enters this phase when all goals have been classified as either unreachable or have been reached.

4.3.4. Externally Used TCL Procedures

This section describes TCL procedures contained in `hvRecipe.tcl` which are intended to be called by the UI process directly. In addition to calling these TCL procedures, the UI calls many native hv commands (e.g. `hvino_create_hnl_design`)

4.3.4.1. `hv_formal_setup` Procedure

The `hv_setup_formal` TCL procedure performs various setup function needed to start a Slave process with intent to subsequently perform formal reachability. After initiating the Slave process, this procedure returns without waiting for the Slave process to complete setting up. Slave progress is tracked via engine messages handled in the various message handler routines.

Currently supported formal engines are either SAT or Symbolic Simulation. The procedure prototype is the following:

```
hv_setup_formal -design <desName> -type <rchType> \
               -dir <wdPath> -log <logName>
```

This procedure starts the hv program in a Slave process, running in the directory specified by `<wdPath>`. If `-log` is specified, all commands issued to the Slave process will be logged in `<logName>`. The caller must specify the design name to be analyzed, and the type of reachability engine to be used: either “SAT” or “SYMSIM”.

After the new channel is opened and a command stack created for it, the Dut and Goal Engines are started for that channel. The Sat or SymSim Engine is also started.

The `hv_dut_setup` procedure is called to setup the Dut Engine. The `hv_load_goals` procedure is pushed on the command stack to be called when a message is received from the Slave saying that the `HvDsg` has been created.

If the Symbolic-Simulator is being setup, the `hv_start_symsim_mgr` command is pushed on the stack to be executed after goals are loaded.

TCL knobs used by this procedure are the following: `hvPath`, `goalFile`, `covVarsFile`, `workDir` (if `-dir` not specified), and `verbMode`.

4.3.4.2. `hv_start_vcs_session` Procedure

This procedure is called after all formal reachability engines have been started to start the VCS explicit state simulator. This procedure will start performing random simulation in the `MODE_INIT` phase described earlier. After random simulation starts, this procedure returns. Procedure prototype is the following

```
hv_start_vcs_session -exe <vcsExe> \
                    -map <mapFile> -root <rootModule> \
                    -dir <wdPath> -log <logName>
```

This procedure starts a new channel, attaches a command stack to it, then executes the compiled VCS simulator in it (given by the path `<vcsExe>`). Depending on knob

settings and use of the `-log` switch, VCS may be started using a number of switches to that program itself.

`<rootModule>` is required and is the verilog module which is being analyzed in the formal engines. `<mapFile>` is required and specifies verilog reg's which are mapped to SEQ elements in the HNL, nets in verilog which are derived clocks in HNL (and therefore have corresponding edge detectors), and the net that is the System Clock in HNL. All names in `<mapFile>` are valid HNL SEQ icell or inet names.

After VCS starts, it performs one reset, it stops and waits for commands (described in a later section of this document). This procedure sends a `$startHV` PLI command to be processed. The `hv_vcs_init_run` procedure is pushed on the command stack to be called when `$startHV` finishes. After these events are initiated, `hv_start_vcs_session` returns without waiting for anything to complete.

`hv_vcs_init_run` is popped from the command stack when VCS has invoked, run a reset sequence and stops. This is detected by a `VCS_STOP` message from VCS, when the VCS engine is in the "initing" state.

TCL knobs used by this procedure are the following: `goalFile`, `workDir` (if `-dir` is not specified), `vcsTestFile`, `outputVcsRchStates`.

4.3.4.3. `hv_terminate_vcs_session` Procedure

This procedure may be called to terminate a test creation session prior to classification of all goals. By default, the test creation initiated by calling the `hv_start_vcs_session` procedure will continue until **all** goals have been classified. In some cases this takes a long time and the user may wish to terminate the process cleanly and save a testcase which reaches only a portion of all goals. This procedure is used for that eventuality. Procedure prototype is:

```
hv_terminate_vcs_session -force <forceLevel>
```

This procedure sends a command to VCS to stop simulating, write out the testcase requested (when `hv_start_vcs_session` was called), and wait before exiting the VCS process. By default, VCS will wait until an in-process formal reach attempt has finished. If `<forceLevel> == 1` is specified, VCS will terminate immediately without waiting.

4.3.4.4. `hv_print_ctg_knobs` Procedure

This procedure is implemented in `infoFile.tcl` where all TCL knobs are defined. This function prints out all current knobs settings. Procedure prototype is:

```
hv_print_ctg_knobs -doc <docRqst>
```

If `<docRqst>` is non-0, then a 1-line description of knob usage will be printed for each knob.

4.3.5. Key Internal TCL Procedures

This section describes in some detail TCL procedures which are used to carry out a CTG session initiated by the earlier externally used procedures. In general, these procedures are not called by the UI.

CTG detects that goals are done being loaded by receiving a HVGOL_ENGINE_WORKING message from the goal engine. Data is "DONE" and the goal engine must be in the "load_goalfile" state.

This procedure uses the TCL knobs: verbMode, goalFile, cicMaxLimit, cicBlkSize, and memInfoFile.

4.3.5.3. hv_do_lmbm_unreach Procedure

This procedure initiates unreachability analysis using LMBM-style fixed point computation. Note that the underlying design may be a partition over the whole machine (true LMBM) or simply a single block which is a subset of the whole machine (not true LMBM). Procedure prototype is:

```

hv_do_lmbm_unreach -init <initFile> -state <vecName> \
    -design <desName> -gfile <goalFile> \
    -gset <goalSet> -dir <wdPath> -log <logName>

```

This procedure opens a new Channel and runs the hv program in Slave mode. The new program is running in directory <wdPath>. A new Command Stack is attached to the Channel, and Dut, Goal, and Image Engines are started for the Channel. The new channel uses <logName> if specified. <goalSet> specifies the name of the GoalSet created by sourcing <goalFile> which is to be checked for unreachable goals. <vecName> is the name of a StateVector created by sourcing <initFile>; this StateVector contains the starting state for fixed point computation.

The following procedures are executed in order for the Channel; this is carried out by pushing the commands onto the Command Stack in reverse order. When each task subsequently finishes, the various message handlers pop the next command off the stack and execute it.

- hv_dut_setup – This procedure loads HNL and creates HvDsg in the Slave process.
- hv_load_goals – This procedure causes goals to be loaded into the Slave process.
- hv_start_lmbm – This procedure initiates the fixed point computation in the Slave process.
- hv_finish_lmbm_unreach – This procedure is called when the fixed point computation completes and uses the fixed point to determine which goals are provably unreachable.
- hv_update_goals – Causes unreachable goals to be communicated from the Slave process to the Master process. The Master subsequently communicates the info to other Slave engines which care.

The hv_dut_setup procedure is directly executed and this procedure returns without waiting for anything to finish. TCL knobs used by this procedure are: hvPath, goalFile (if -gfile not used), covVarsFile, workDir (if -dir not used), verbMode.

4.3.5.4. hv_start_lmbm Procedure

This procedure is called to initiate pseudo-LMBM style fixed point computation using symbolic state traversal techniques. This fixed point is subsequently used to prove

unreachable goals. This procedure is called after the DUT and Goals have been properly setup in the Slave process. Procedure prototype is:

```
hv_start_lmbm -chnl <chnlId> -init <initFile> \  
-state <vecName>
```

<chnlId> is the Channel which has had DUT created and Goals loaded properly (as described by previous sections). <vecName> is the name of a StateVector created by sourcing <initFile>; this StateVector contains the starting state for fixed point computation.

Note that current usage does not really do LMBM type fixed point computation because the underlying machine is not partitioned into submachines. If the underlying machine and HvCic were partitioned, the state traversal would be true LMBM. This procedure issues the following commands to the Slave process without waiting for any of them to finish:

- hvflc_create_cover
- hvfml_create_lib
- hvfac_create_rpb_from_hbv_state – The starting state <vecName> is converted to an RPB for use by the Image Computation algorithms.
- hvfmm_compute_lmbm – This procedure actually computes the fixed point. The Image Engine is set to the state “run_lmbm”. The resultant fixed point name is “lmbm_fpoint”.

The completion of this procedure is detected by receiving a IMG_ENG_LMBM_DONE message from the Image computation engine. A command is popped from the stack when this message is received.

This procedure uses the TCL knobs: verbMode.

4.3.5.5. hv_finish_lmbm_unreach Procedure

This procedure is called after fixed point computation has finished.. The procedure prototype is:

```
hv_finish_lmbm_unreach -chnl <chnlId> \  
-lmbm <lmbmName> -gset <goalSet>
```

The Master process sends commands to the Slave of Channel <chnlId> to cause unreachability computation to occur for the goals in <goalSet> using the fixed point <lmbmName>.

The following commands are sent to the Slave process of <chnlId>:

- hvfac_print_rpb – This is RPB-specific information about the fixed point RPB named “lmbm_fpoint”.
- hvfac_count_minterms_in_rpb – This command projects lmbm_fpoint onto the coverage variables contained in InetSet CovVars, then prints some statistics.

- `hvgol_report_unreachable` – This command makes the fixed point known to the appropriate GoalSet. The number of Reachable, Unreachable, and Unknown goals are printed in a message for consumption by the Master process.

At the completion of this procedure (more specifically, the resultant message from the `hvgol_report_unreachable` command), the Master process knows how many goals are unreachable, but not which ones. The `hv_update_goals` procedure describes how this task is completed.

The completion of this procedure is detected by receiving a `HVGOL_CMD_POP` message with data value “report_unreach”. A command is popped from the stack when this message is received.

This procedure uses TCL knobs: `verbMode`.

4.3.5.6. `hv_update_goals` Procedure

This procedure is called after a Slave process has determined unreachable goal information. The goals of the Master process must be updated and this procedure initiates the update. Procedure prototype is:

```
hv_update_goals -chnl <chnlId> -gset <goalSet>
```

This procedure sends a `hvgol_write_update_info` command to the Slave process of Channel `<chnlId>`. The Goal Engine for that Channel is set to the “update_goal” state and the procedure returns.

The completion of the `hvgol_write_update_info` command is detected by receiving a `HVGOL_GOALSET_WRITE_UPD` message. When this message is received, the update file is sourced in the Master process so that goals in the Master are correct. A command is then popped from the command stack and executed.

This procedure uses TCL knobs: `verbMode`.

4.3.5.7. `hv_vcs_initial_run` Procedure

This procedure is called when VCS has finished invoking and running its first reset sequence. The procedure prototype is:

```
hv_vcs_initial_run -chnl <chnlId> -gfile <goalFile>
```

This procedure is given the VCS Channel via `<chnlId>`. `<goalFile>` is a sequence of calls to the PLI routines `$createSignalSetHV`, `$addToSignalSetHV`, and `$defineFsmCoverHV` which cause creation of goalsets which are compatible with the goalsets created in the formal engines.

The following commands are sent to VCS:

- `source <goalFile>` – This causes the necessary goals to be created in VCS
- If requested, pre-reached coverage states may be loaded into VCS by calling the `$readCoverInfoHV` PLI routine. These pre-reached goals are identified by use of the `inputVcsRchStates` TCL knob.

- The \$configStopPointHV PLI routine is called to setup stop conditions for VCS.
- The hv_vcs_stopped command is pushed onto the Command Stack for the VCS Channel. The VCS Engine of the Channel is set to the “running” state.
- A run command is sent to VCS. This is the famous “.” command.

The procedure then returns without waiting for completion of the run command. Completion of the initial run is detected by a receiving a VCS_STOP message when vcs is in the “initing” state. Completion of non initial runs is detected by a VCS_STOP message while in the “stop_pending” state.

This procedure uses TCL knobs: verbMode, goalFile, inputVcsRchStates, initCycles, initRunsCnt.

4.3.5.8. hv_vcs_stopped Procedure

This procedure is called when VCS has stopped after running biased random simulation. This procedure examines various pieces of information to determine what should be done next. Procedure prototype is:

```
hv_vcs_stopped -chnl <chnlId>
```

VCS has stopped running on <chnlId>. The following decision process takes place, depending on CTG analysis phase:

- **MODE_INIT** – This is the mode where CTG is reaching as many goals as reasonably possible from the reset state using VCS. If the number of VCS runs from the reset states is less than initRunsMax, or if formal reachability is inhibited (using inhibitFormalReach), then 1) tell VCS to perform a reset, 2) push hv_vcs_stopped onto the VCS command stack, 3) initiate a VCS run, 4) return.

If the number of initial random runs is greater than initRunsMax then a formal search is initiated from the reset state. CTG analysis phase is set to MODE_INIT_SAT (if satCycMax is > 0), else phase is set to MODE_INIT_SYM). Following this phase adjustment, the following steps are taken: 1) tell VCS to perform a reset then stop in the reset state, 2) push hv_run_formal onto the VCS command stack, 3) tell VCS to run, 4) tell VCS to write state to a file using \$reportDutStateHV, 5) tell VCS to write newly-reached goals to file using \$reportNewCoverDataHV, 6) return.

An additional decision process in this mode is to choose when unreachability analysis is initiated. Unreachability is launched and runs concurrently with reachability. Unreachability is launched from the same reset state as the first formal search, or earlier if unreachInitCnt is non-0. This knob requests that the unreachInitCnt-th reset state will be used to launch unreachability. If unreachability is launched, then the hv_do_lmbm_unreach procedure is called and VCS is told to write state into the file ‘lmbmInit.tcl’ using the \$reportDutStateHV PLI routine. The VCS engine state is set to ‘reach_lmbm_after_reset’ in order for the message handler to track what is going on.

- **MODE_INIT_SAT, MODE_INIT_SYM** – This is the phase where CTG is reaching as many goals as possible using SAT/SymbolicSim from the reset state. VCS has stopped at some non-reset state and a formal search is to be setup and kicked off from the reset state. The following actions are taken: 1) tell VCS to perform a reset then stop in reset state, 2) push hv_run_formal onto the VCS command stack, 3) tell VCS to run, 4) tell VCS to write state to a file using \$reportDutStateHV, 5) tell VCS to write newly-reached goals to file using \$reportNewCoverDataHV, 6) return.
- **MODE_SAT, MODE_SYMSIM_SAT, MODE_SYMSIM** – In these phases, CTG is using formal searches from deep states reached using VCS. At this point, VCS has stopped at one of these deep states and this routine will either kick off a formal search, or reset VCS and try to reach more interesting deep states. The following actions are taken (in pseudo-code form):

```

If ((stopped due to cycleLimit) &&
    (cycleLimitMax <= cycleLimitStopCnt))
    push hv_vcs_stopped onto the vcs command stack
    tell VCS to reset
    tell VCS to run
else
    if (phase is MODE_SAT)
        push hv_run_formal -type SAT on VCS stack
    else if (phase is MODE_SYMSIM)
        push hv_run_formal -type SYM on VCS stack
    else
        if (last formal was SAT)
            push hv_run_formal -type SYM
        else
            push hv_run_formal -type SAT
        endif
    endif
    tell VCS to dump state ($reportDutStateHV)
    tell VCS to dump new goals ($reportNewCoverDataHV)
endif

```

- **MODE_DONE** – VCS has reached all requested goals, and CTG needs to update the goals in the Master process then halt cleanly. The following actions are taken: 1) push hv_run_formal onto the VCS command stack, 2) tell VCS to dump state using \$reportDutStateHV, tell VCS to dump all recently reached goals using \$reportNewCoverDataHV.

When the \$reportNewCoverDataHV PLI routine is issued, completion is detected by receipt of a VCS_NEW_WRITTEN message from VCS. A command is popped from the stack in response to this message.

This procedure uses TCL knobs: verbMode, initRunsMax, inhibitFormalReach, updateFile, stateFile, resetConstFile, runConstFile, satCycMax, unreachable, stateVector, goalFile, workDir, design, goalSetNames, cycleLimitMissMax.

4.3.5.9. hv_vcs_replay_done Procedure

This procedure is called when VCS has completed replaying a trace generated from a formal search engine. Procedure prototype is:

hv_vcs_replay_done -chnl <chnlId> -action <actType>

<actType> specifies which action the initiator of the replay thought might be appropriate; must be "RUN" or "BACK2BACK".

- **RUN** – VCS is to run for a period of time using biased random simulation. The hv_vcs_stopped procedure is pushed onto the VCS command stack, and VCS is then told to run.
- **BACK2BACK** – This action is to use "Back-to-Back" formal searches. In this case, random simulation is not allowed to run immediately after a formal search has succeeded in hitting a new goal. hv_run_formal is pushed onto the VCS command stack. VCS is told to 1) write its state to file by \$reportDutStateHV, then 2) write recently reached goals to file by calling \$reportNewCoverDataHV

This procedure uses the TCL knobs: verbMode, updateFile, stateFile, resetConstFile.

4.3.5.10. hv_start_symsim_mgr Procedure

This procedure is called to setup the Symbolic Simulation manager prior to performing any searches using that engine. This procedure is called once at the beginning of a session, after goal and dut engines are properly set up, Ord, Sth, and Mch are already built, etc. Typically called from hv_setup_formal. Prototype is:

hv_start_symsim_mgr -chnl <chnlId>

<chnlId> is the channel that should have a Symbolic Simulation manager built. The hvprn_sim_manager create command is called in the child process. It is expected that hvprn_\$design, hvflc_\$design, and hvord_\$design are valid Mch, Flc, and Ord objects, respectively.

Completion of this sequence is detected by receipt of HVSYM_MGR_SETUP message. A command is popped from the command stack in response to this message.

4.3.5.11. hv_run_formal Procedure

This procedure is called to initiate a formal search on a properly opened and set up Channel. Function prototype is:

hv_run_formal -chnl <chnlId> -sfile <startFile> \
-update <updateFile> -const <constFile>
-type <rchType>

<rchType> must be "SAT" or "SYMSIM", depending on which engine should be used for the formal search. <startFile> is a TCL file to be sourced in the Slave process to create a StateVector; this StateVector defines the starting state for the search engine. <updateFile> is a TCL file to be sourced in the Slave and Master processes to update the GoalMgr with states reached by VCS since the last <updateFile> was generated. <constFile> is a TCL file to be sourced in the Slave process to cause the search engine to see certain primary inputs as of a constant value. Note: This <constFile> is expected to be obsoleted when HNL-based environmental modeling can specify the same data.

Actions carried out in this procedure are:

1. Issue commands to all Slave processes running symsim or sat engines to "source <updateFile>". This causes all processes to have full knowledge of what goals have been reached by VCS.
2. Push hv_formal_stopped onto the Command Stack of the channel to perform the formal search.
3. If we are initiating back-to-back searches, adjust resource limits accordingly. If <rchType> is "SAT", iteration limit is set to satB2BCycMax or satCycMax. If <rchType> is "SYMSIM", iteration limit is set to symsimB2BCycMax or symsimCycMax.
4. If <rchType> is SYMSIM, source <constFile> in the child process, then source \$SYNOPSIS/auxx/ctg/tcl/runPrm.tcl in the Slave. If <rchType> is SAT, then call the hvcnf_sat command in the child process.
5. The procedure then returns.

Completion of the formal search is detected by receipt of either HVSYM_SIM_DONE or HVSAT_REACH_DONE messages. A command is popped from the command stack in response to either of these messages.

This procedure uses the following TCL knobs: symsimCpuMax, symsimCycMax, symsimB2BCycMax, satCpuMax, satCycMax, satB2BCycMax, goalSetNames,

4.3.5.12. hv_formal_stopped Procedure

This procedure is called when a formal search engine has terminated a search. The search may have terminated after finding one or more goals, or it may have terminated after exceeding some user-provided resource limits. Procedure prototype is:

```
hv_formal_stopped -chnl <chnlId> -type <rchType>
```

<rchType> is either "SAT" or "SYMSIM" and is the type of search engine which was initiated on the Channel of <chnlId>.

Note that when the search engine completed, it reported how many goals were found; this data recorded in either satHandler.tcl or symsimHandler.tcl.

The decision process of this routine is outlined by the following pseudo-code.

```
If (phase is MODE_INIT_SYM)
  If (search found 0 goals)
    If (satCycMax>0)
      Set phase to MODE_SAT
    Else
      Set phase to MODE_SYMSIM
    Endif
    Set non-init-phase VCS stop conditions \
      & freshness using $configStopPointHV
    push hv_vcs_stopped on VCS cmd stack
    set VCS engine state to "running"
    tell VCS to run
  Else
    Push hv_vcs_replay_done on VCS cmd stack
    Tell VCS to bias weights
```

```

        Tell VCS to replay a trace
        Set VCS engine state to "running"
        Tell VCS to run (do the replay)
    Endif
Elseif (phase is MODE_INIT_SAT)
    If (search found 0 goals)
        If (symsimCycMax>0)
            set phase MODE_INIT_SYM
        Else
            set phase MODE_SAT
        Endif
        If (phase is MODE_INIT_SYM)
            Call hv_run_formal
        Else
            Set non-init-phase VCS stop conditions \
                & freshness using $configStopPointHV
            push hv_vcs_stopped on VCS cmd stack
            set VCS engine state to "running"
            tell VCS to run
        Else
            Push hv_vcs_replay_done on VCS cmd stack
            Tell VCS to bias weights
            Tell VCS to replay a trace
            Set VCS engine state to "running"
            Tell VCS to run (do the replay)
        Endif
Elseif (phase is MODE_SAT, MODE_SYMSIM_SAT, MODE_SYMSIM)
    If (search found 0 goals)
        If (phase is MODE_SAT)
            If (symsimCycMax>0)
                set phase MODE_SYMSIM_SAT
            Endif
        Elseif (phase is MODE_SYMSIM_SAT)
            If (lastFormalEng was SAT)
                Set phase to MODE_SYMSIM
            Endif
        If (#consecMisses > formalMissThresh)
            Push hv_vcs_stopped on VCS stack
            Set #consecMisses to 0
            Tell VCS to reset
        Elseif (lastFormalEng was SYMSIM)
            Push hv_vcs_replay_done on VCS stack
            Tell VCS not to bias weights
            Tell VCS to replay a trace
        Else
            Push hv_vcs_stopped on VCS stack
        Endif
    Else
        If (#consecHits > formalClearThresh)
            Tell VCS to clear bias weights
        Endif
        If (back2back enabled)
            Push hv_vcs_replay_done on VCS stack \
                (with BACK2BACK action)
        Else
            Push hv_vcs_replay_done on VCS stack \

```

```

                                (with RUN action)
        Endif
        Tell VCS to bias weights
        Tell VCS to replay
    Endif
    Set VCS engine state to "running"
    Tell VCS to run
Endif

```

This procedure uses the following TCL knobs: `verbMode`, `satCycMax`, `symsimCycMax`, `freshStartFrac`, `freshSampFrac`, `rsimCycles`, `freshSuppress`, `updateFile`, `stateFile`, `resetConstFile`, `formalMissThresh`, `formalClearThresh`, `backToBackEnabled`.

4.3.5.13. `hv_update_cover_metric` Procedure

This procedure computes the actual cover metric used to determine progress made in the overall coverage session. This is a function of all goals/goalsets currently defined. Sets the `covMetric` TCL variable. Procedure prototype is:

```

hv_update_cover_metric

```

This procedure takes no arguments.

4.3.6. TCL Knobs and Variables

The CTG control mechanisms, heuristics, and decision algorithms are largely implemented in TCL code running in the Master process. This TCL code allows for many user-specifiable configuration settings; these are called TCL “knobs”. This control TCL code also needs to keep track of session progress and status; this makes use of TCL “variables” which change as the session progresses.

The TCL language is limited in variable scope and data types. Variables are visible either globally or local to a single procedure. Global variables used in a procedure must be declared in that procedure as ‘global’ before use in the procedure. TCL variables must be of type string or array; array variables are indexed by strings and are one-dimensional.

All TCL knobs and variables are organized as elements of a single global array variable named “`ctgInfo`”. The previously mentioned knob “`satCycMax`”, for example, is actually referenced as “`ctgInfo(satCycMax)`”. When a TCL knob or global variable is added to the `ctgInfo` array, a documentation entry is also entered in a corresponding array “`ctgInfoDoc`”. The documentation line for `ctgInfo(satCycMax)` is thus contained in `ctgInfoDoc(satCycMax)` and its value is: “Maximum SAT cycles allowed per search”. A TCL procedure `hv_print_ctg_knobs` is provided to print knob values and documentation; this procedure was described earlier.

This remainder of this section documents knobs and variables in use at the time of writing this document.

4.3.6.1. TCL Knobs

The following knobs are used to configure the CTG TCL control code. In each case, the knob name is provided without the “`ctgInfo()`” for clarity. Knob default values are also provided, where relevant.

- `workDir "."` - Directory that child process are started in, by default.

- **stateFile** "stateFile.tcl" - Name of temporary file VCS uses to write current design state into. When sourced, a StateVector is created then each state element is set to the current value (0/1) in the VCS simulation session.
- **stateVector** "StateVec" - Name of State Vector that updateFile.tcl saves state in when it is sourced
- **updateFile** "updateFile.tcl" - Name of temporary file VCS uses to store reached states in when this info is shared with other processes. When sourced, all goals and goalsets are updated with goals which have been reached since the last time VCS was asked to report newly reached states.
- **runConstFile** "<no_consts>" - Name of file which can be sourced to set certain signals to constant values during symbolic simulation. This file consists of a series of hvprm_input_set_to_constant commands
- **resetConstFile** "<no_consts>" - Set certain signals to constant values for Symbolic Simulation from the reset state. Similar to runConstFile, except used only when simulating from reset state.
- **goalFile** "goalFile" - Basename of files used to create goals in hv and VCS child processes. It is expected that <goalFile>.vlog can be sourced in VCS, and <goalFile>.tcl can be sourced in hv.
- **goalSetNames** "<no_goalsets>" - Space-delimited list of GoalSet names which are created by sourcing the above <goalFile>.
- **goalNames** "<no_goals>" - Space-delimited list of Goal names which are created by sourcing the above <goalFile>.
- **covVarsFile** "covVars.tcl" - Name of file to source in hv which creates an InetSet named 'CovVars'. This InetSet contains all coverage variables used in all goals and goalsets analyzed in this session.
- **unreach** "UNREACH_NONE" - Knob which specifies style of unreachability analysis to perform. Valid values are:
 - UNREACH_NONE: No unreachability used
 - UNREACH_LMBM: Use lmbm-type fixed point computation
- **unreachInitCnt** 3 - Specify which reset state should be used to initiate unreachability from. In practice, not all verilog registers in a design are set to a specific value by the reset sequence. Thus, sometimes X logic values may take many simulation cycles and/or resets to flush from the system. In these cases, the first reset state generated may not be a good start-point for unreachability analysis. This knob specifies that the i-th reset state should be used for unreachability computations. Unreachability will start no later than when the first formal search is initiated. I.E. when MODE_INIT is exited.
- **inhibitFormalReach** 0 - If this knob is non-0, then formal search mechanisms are not used. CTG stays in MODE_INIT forever.

- **verbMode 1** - Specifies verbosity mode to use in TCL control code running in the Master process. Set to one of:
 - **ctgInfo(verbUSER)**: Not much verbosity
 - **ctgInfo(verbAPPLICATION)**: a little more
 - **ctgInfo(verbDEVELOPMENT)**: still more
 - **ctgInfoDoc(verbPANIC)**: so much we get lost
- **freshMode "START_FRESH_OFF"** - This knob specifies when freshness detection should start to be used. Valid values are:
 - **START_FRESH_OFF**: dont use freshness.
 - **START_FRESH_DYNAMIC**: use freshness after the first miss
 - **START_FRESH_ALWAYS**: freshness on from the very start
- **freshSuppress 3000** - Number of cycles to suppress freshness detection at the beginning of a random simulation run
- **freshSampFrac 1000** - Sample fraction, in 1/1000 percent units, to use for freshness detection. If a given state has been seen less than this percentage, then that state is said to be "fresh".
- **freshStartFrac 10000** - Start fraction used for freshness detection. When VCS is asked to save current state to a file, this is taken to be a start point for a formal search. If a given state has been used as start point less than this percentage of all times a start has been initiated, then the state is said to be fresh.
- **cicBlkSize 50** - The size of blocks to use for HvCic object used for unreachable analysis.
- **cicMaxLimit 3000** - The maximum number of latches to use in the HvCic object used for unreachable analysis.
- **rsimCycles 10000** - The number of clock cycles VCS will perform random simulation before stopping for direction. VCS runs this many clock cycles past the time the last new state (i.e. goal) was reached. The TCL control code in the Master process then determines if the current state should be used for formal search, if a reset should occur, etc.
- **initCycles 2000** - The number of clock cycles of random simulation to perform after a reset sequence. VCS will run this many cycles past the last new state (i.e. goal) seen
- **initRunsMax 2** - How many sequences of Reset+RandomSim will be performed before CTG exits MODE_INIT

- **dynamicBiasCycles** 1000 - How many cycles of dynamically biased random simulation will be run after a successful formal search to hit a new goal. After this many cycles, user-biased random simulation will be used.
- **satCpuMax** 600 - Maximum number of CPU seconds which is allowed for a formal search using SAT.
- **satCycMax** 50 - Maximum number of simulation steps (i.e. clocks) which are allowed for a SAT formal search
- **satB2BCycMax** 10 - Maximum number of simulation steps allowed for a SAT search when used as the second search in a BackToBack search. Generally this number is less than satCycMax.
- **symsimCpuMax** 600 - Maximum number of CPU seconds which is allowed for a formal search using Symbolic Simulation.
- **symsimCycMax** 50 - Maximum number of simulation steps (i.e. clocks) which are allowed for a Symbolic Simulation formal search
- **symsimB2BCycMax** 25 - Maximum number of simulation steps allowed for a Symbolic Simulation search when used as the second search in a BackToBack search. Generally this number is less than symsimCycMax.
- **formalReplayFile** "prm_sequence" - Name of temporary file used by formal search engines to save replay traces in. This file is then read by VCS to replay the trace
- **cycleLimitMissMax** 2 - This knob specifies how many times a formal search may be initiated from a non-fresh state when a miss occurs. The idea is that often a search from non-fresh state has no hope of succeeding. Note that these searches occur when rsimCycles of simulation have occurred in VCS since that last goal was hit.
- **backToBackEnabled** "OFF" - This knob specifies if BackToBack should be enabled or not. Values of OFF and ON are allowed. BackToBack causes a formal search to be performed immediately after a formal search has succeeded in hitting a new goal. The default behavior is to perform dynamically biased random simulation from these points.
- **formalMissThresh** 2 - Number of consecutive formal searches which may fail before VCS is instructed to perform a reset.
- **formalClearThresh** 3 - Number of consecutive formal searches which may succeed before dynamic biasing weights are cleared in VCS.
- **vcsTestFile** "<no_testfile>" - Name of file which the generated test is to be saved in. This test may be used later as a stand-alone regression test in VCS. Two files are saved:
 - <testFile>.ctg contains the control program to reproduce the input stimulus.
 - <testFile>.vgol specifies the goals and goalsets used for the test.

- **inputVcsRchStates** "<no_inputfile>" - This file specifies goals which are considered to have been previously reached. These goals are classified as "reached" at the beginning of the CTG session, and no attempt is made to generate a trace to these goals.
- **outputVcsRchStates** "" - This file specifies a file to record goals which are reached during this analysis session. This file format is suitable as input to later CTG sessions.

4.3.6.2. Variable Usage

This section lists global variables used by multiple TCL procedures in the TCL control code running in the Master process.

- **ctgMode** - This variable contains which phase CTG is currently in. One of MODE_INIT, MODE_INIT_SAT, MODE_INIT_SYM, MODE_SAT, MODE_SYMSIM_SAT, MODE_SYMSIM, or MODE_DONE
- **unreachStarted** - Variable that says unreachability has been started (if non-0)
- **initRunsCnt** - Number of MODE_INIT simulation runs which have been performed from the reset state.
- **covMetric** - This variable is the number of goals left to be classified
- **formalMissCnt** - Number of consecutive formal search misses seen.
- **formalHitCnt** - Number of consecutive formal search hits seen.
- **formalTraceCnt** - How many goals the last formal search found
- **lastReachType** - Which type formal search engine was used last
- **back2BackInProgress** - A BackToBack search is in progress
- **lastVcsStopReason** "<no_reason>" - Reason that the last VCS simulation stopped. One of VCS_STOP_UNKNOWN, VCS_STOP_DONE, VCS_STOP_CYCLE_LIMIT, VCS_STOP_FRESH, VCS_STOP_END_REPLAY
- **cycleLimitMissCnt** - How many formal search misses have when seen when at cycle limit stop point.
- **hvPath** "/vobs/propver/src/hv/bin-gccsparcOS5/hv-g" - Pathname to where hv executable lives

4.3.6.3. ctgVcsCoverObjs Variable

When a coverage object is created in VCS (e.g. creating a GoalSet by using the \$defineFsmCoverHV PLI routine), it is given a unique "handle". This handle is then used as input to other PLI routines during the CTG session. The same coverage object is known by name in the Master process and other hv Slave process. The ctgVcsCoverObjs variable is an array that maps a coverage object name (e.g. the GoalSet name) to its

corresponding handle in the VCS process. For example: `ctgVcsCoverObjs(GoalSet1) = 1`.

4.3.6.4. `ctgVcsCoverStatus` Variable

The `ctgVcsCoverStatus` variable is an array which records current classification counts for the goals which correspond to a coverage object in VCS. Index into the array is the VCS handle of the object. Data value is a string of 3 space-delimited numbers: `ctgVcsCoverStatus(1) = "#unknown #unreachable #reached"`.

5. Goal Creation and Usage

The creation, maintenance, and use of goals is central to the CTG methodology. Goals are, after all, what are being "covered", "reached", "proven", "classified", etc. The hv program provides a number of goal-related TCL commands which are documented in this section. In addition, the PLI code linked into VCS provides certain goal-related capabilities which are discussed.

5.1. Introduction

In general, each of the Slave processes in use in the CTG system will need some or all of the goal information collected by the other Slave processes. The TCL control code running in the Master process is responsible for updating the necessary goal information in a Slave process before asking for an analysis by that Slave.

Note that in many cases a Slave does not need all accumulated goal information before carrying out its portion of the CTG session. For example, unreachability analysis performed by LMBM fixed point computation has no need to know that some goals have already been reached.

At all times, the Master process is the central keeper of the union of all goal results determined by all child processes. The Master process TCL code is responsible for propagating all generated results from Slaves to the Master, and from the Master to Slaves that need the information.

5.2. HvGol Package TCL Commands

The HvGol package provides the following TCL commands which are used to manage goal information within the CTG system. Note that not all `hvgol_*` commands are used currently by CTG. Only those commands in use or otherwise of interest are mentioned here.

- `hvgol_create_goal` - This command creates a goal from a single signal or an `HbvStateVector`. The goal is given a name at creation time.
- `hvgol_create_goalset` - This command creates a goalset given an `InetSet` which contains the coverage variables of interest. Given N coverage signals which may take binary logic value $B=\{0,1\}$, then $2^{**}N$ goals are implied by the `GoalSet` and have all logic values given as $B^{**}N$.
- `hvgol_goalset_add_info` - This command is used to incrementally specify reached/unreachable state information for an existing `GoalSet`. Unreachable states must be represented as either fixed point `HvRpb` or as a `HvHbv StateVector`. Reached states are specified as `HvHbv StateVector`.

- `hvgol_goalset_relations` - This command copies out some or all of the `HvSth_Relation_t`'s which are used internally to represent the Unknown, Unreachable, and Reached goals of the GoalSet. The user provides the names used to create named HvMgr instances of type `HvSth_Relation_t`.
- `hvgol_write_update_info` - This command writes into a file the `HvSth_Relation_t` objects which represent the unknown, unreachable, and reached relations for a given GoalSet. This information is suitable for reading in using the `hvgol_read_update_info` command.
- `hvgol_read_update_info` - This command reads goalset information written by one hv session, and used in another. The provided reached, unreachable, and unknown Relations from one goalset are used to update the same GoalSet with new information.
- `hvgol_report_goal` - This command reports various information about a Goal.
- `hvgol_report_goalset` - This command reports various information about a GoalSet
- `hvgol_report_unreachable` - This command reports reached/unreachable state information to a goal or goalset. This command is somewhat mis-named, as the semantics are that the caller is reporting to the Goal, rather than vice-versa. Unreachable states are in RPB fixed point form. Unreachable states are specified as an `HvHbv StateVector`.

5.3. HvPli Services For VCS Usage

Certain goal capabilities are implemented in the VCS Slave process using standard PLI capabilities. VCS is invoked in interactive mode so that these PLI routines may be "called" by the Master process. The PLI routines provided for goal creation/maintenance are the following:

5.3.1. createSignalSetHV PLI Routine

```
$createSignalSetHV(" <setName> ")
```

This PLI routine is used to define a SignalSet in the VCS process. A SignalSet is entirely analogous to an InetSet in one of the hv processes. The "setName" argument is a string that is expected to correspond to a InetSet in the Master process.

5.3.2. addToSignalSetHV PLI Routine

```
$addToSignalSetHV(" <setName> ", <signal>, " <addMode> ")
```

This PLI routine is used to add a Verilog signal to the named SignalSet. `<signal>` may be either a net or register in verilog. `<addMode>` is a string that is either PRE or POST to indicate the signal should be prepended or appended to the SignalSet. The order of the signal in the SignalSet is expected to be the same as the order of signals in the corresponding InetSet.

5.3.3. reportSignalSetHV PLI Routine

```
$reportSignalSetHV(" <setName> ", " <fileName> ", " <fmt> ")
```

This PLI routine writes a SignalSet given by <setName> out to <fileName> in a specified <format>. <format> is a string that is either "HV" or "PLI". These formats are suitable to be sourced in either hv or VCS programs to reconstruct the SignalSet.

5.3.4. defineFsmCoverHV PLI Routine

```
$defineFsmCoverHV("<objName>", "<signalSetName>", \
    <sysClk>, <offSet>, <fsmHandle>)
```

This PLI routine creates an Fsm type coverage object named <objName>. The goals are defined as all [1,0] combinations of the signals contained in <signalSetName>. If there are N signals in SignalSet, then 2*N individual goals are defined with this command. These signals are sampled relative to the positive edge of the clock signal <sysClk>. <offSet> specifies a non-negative offset by which the sample time can be adjusted so that the sample happens a short time after the positive edge of <sysClk>. This routine allocates a unique positive integer handle for the coverage object; this is returned as <fsmHandle>.

5.3.5. defineBitToggleCoverHV PLI Routine

```
$defineBitToggleCoverHV("<objName>", "<signalSetName>", \
    <sysClk>, <offSet>, <fsmHandle>)
```

This PLI routine creates a Toggle coverage object named <objName>. Signals given by the <signalSetName> are to be toggled to both 1 and 0; these are the goals. If there are N signals in SignalSet, then 2*N individual goals are defined with this command. These signals are sampled relative to the positive edge of the clock signal <sysClk>. <offSet> specifies a non-negative offset by which the sample time can be adjusted so that the sample happens a short time after the positive edge of <sysClk>. This routine allocates a unique positive integer handle for the coverage object; this is returned as <fsmHandle>.

5.3.6. getCoverMetricHV PLI Routine

```
$getCoverMetricHV(<covHandle>, <covNumber>)
```

This routine determines how many goals remain unclassified in the coverage object identified with <covHandle>. If <covHandle> is -1, then all currently-defined coverage objects are used and the returned <covNmuber> is the summation of all unclassified goals.

5.3.7. reportNewCoverDataHV PLI Routine

```
$reportNewCoverDataHV(<covHandle>, "<fileName>")
```

This PLI routine writes to <fileName> all goals which have been covered since the last call to this routine. <fileName> must be writeable, and will be overwritten by this routine. The output file is expected to be sourced by the hv program to update the corresponding GoalSets in the hv process. File contents are the following:

- create a StateVector with the same name as the GoalSet (hvhbv_new_vector -init DC ...)
- Set each care bit in the StateVector using hvhbv_set_vector_bit
- Update the GoalSet using hvgol_goalset_add_info to present the properly loaded StateVector as a new reached state for the GoalSet.

5.3.8. informUnreachCountHV PLI Routine

```
$informUnreachCountHV(<covHandle>, <#unreach>)
```

This PLI routine is used to inform the PLI code that a certain number of states of the given coverage object have been proven unreachable. The effect of this routine is to adjust the coverage metric for the coverage object given by the handle <covHandle>. Note that this command is additive for a given coverage object; this allows for unreachable goals to be proven throughout an analysis session by different unreachable algorithms.

5.3.9. printCoverInfoHV PLI Routine

```
$printCoverInfoHV(<covHandle>, "<fileName>", "<fmt>")
```

This PLI routine writes out reached state coverage information out to the file named <fileName>; this file must be writeable and will be overwritten. If <covHandle> is -1, then all current coverage objects are written, else the one specified by the handle <covHandle>. <format> is a string and is one of "COMPACT", "COMPACT_REACH", or "HUMAN". "COMPACT" and "COMPACT_REACH" formats are intended to be readable using the \$readCoverInfoHV PLI routine. "COMPACT" simply writes out the cover object name and coverage signals. COMPACT_REACH writes cover object name, coverage signals, and reached states. "HUMAN" writes information out in a human-readable form.

5.3.10. readCoverInfoHV PLI Routine

```
$readCoverInfoHV("<fileName>", <use_rch>)
```

This PLI routine reads coverage object definitions from the file given by <fileName>. <use_rch> is an integer; if non-0, then any reached states contained in <fileName> are taken to be previously-reached states and are immediately considered as reached. The next call to \$reportNewCoverDataHV will report these states as reached.

5.4. Usage Scenarios

When goals are determined to either be unreachable or reached, this information must be communicated to other parts of CTG which need this information. This section describes these system-level interactions.

5.4.1. Unreachable State Computation

When unreachable analysis has been completed by one of the Slave processes, the unreachable state data must be reported to the Master process, and conveyed to the VCS process so that the coverage metric can be adjusted properly. The Slave process is told to write unreachable states to file using the hvgol_write_update_info. The Master process then uses the hvgol_read_update_info command to update its central copy of the GoalSet information. The number of unreachable states is reported to the VCS Slave by calling the \$informUnreachCountHV PLI routine.

If a SAT solver is active, then the unreachable states are also read into that process.

5.4.2. Reached State Information

The CTG system does not consider a Goal to have been reached until VCS has generated an input sequence that actually reaches a state which covers the Goal. When a Goal has

been reached, then this information must be conveyed to the Master process, as well as any Slave processes which can take advantage of this information.

This update process occurs each time VCS stops simulation and waits for input directive from the Master process. The following actions then occur:

- VCS is told to write out newly reached goals using the \$reportNewCoverDataHV PLI routine.
- The Master process updates its copy of the GoalSet by sourcing the file of update information. As previously described, this file creates a vector of Don't-Care bits then sets all care bits to the appropriate value. This StateVector is then used with the hvgol_goalset_add_info command.
- All Slaves used for reachability analysis are also told to source the update file.

6. UI Interface

The User Interface of Figure 2 interacts with the core CTG system via the Master process. As previously described, commands are issued to the Master process via a unix pipe which feeds stdin of the Master process. All native hv commands, as well as the externally usable TCL control routines in hvRecipe.tcl, are available for UI needs. This interface allows the UI to be written in either Graphic or Textual modes without any changes, from the Master process perspective. Refer to some other UI-specific document for detailed issues about User-level interactions with CTG.

7. VCS Interface

As indicated earlier, the explicit state simulation engine integrated into CTG is based on VCS. CTG capabilities are linked into VCS directly via PLI techniques for direct simulation access, and using VERA mechanisms for testbench control, constraints, and biasing.

For detailed descriptions of VERA coding and compilation issues for use with CTG, refer to appropriate CTG VERA documentation. This section describes the specific control & interface mechanisms for CTG interactions with VERA code.

For more detailed VCS and PLI descriptions, refer to appropriate product literature. This section document CTG interfacing and control mechanisms present in the VCS Slave process.

7.1. PLI Interfacing – hvPli

As part of a product deliverable, CTG provides PLI routines embodied in a precompiled hvPli.a archive library which is linked to the VCS compiled simulator. A hvPli.tab interface file is provided so that these PLI routines can be called from the user PLI code, as well as interactively from the VCS interactive command line.

PLI routines used in the CTG product are the following. Note that Goal-specific PLI routines were described earlier in this document and are not repeated here.

7.1.1. startHV PLI Routine

```
$startHV(<rootInst>, "<mode>", "<regMapFile>", \
        "<inGoalFile>", "<outGoalFile>")
```

This PLI routine is called at the very start of a CTG session to properly setup and initialize the CTG PLI mechanisms. This routine is called either by the Master process when creating a testcase, or by the CoverBooster program when replaying a previously-created testcase.

<rootInst> is a module in the verilog netlist which is being formally analyzed.

<mode> specifies whether a test is being created ("SLAVE_CREATE") or a previously-created test is being replayed ("FREE_REPLAY").

<regMapFile> is a file which specifies the HNL name of sequential elements in the formally-analyzed design. These names are decoded, together with <rootInst>, to identify registers in the verilog which comprise "state" of the Design Under Test. This information is especially relevant to the \$reportDutStateHV PLI routine. Only registers which are relevant to the HNL model (i.e. those verilog registers "inferred" as sequential elements by the synthesis process) are reported as DUT state for use as start point of formal searches. <regMapFile> is required for mode "SLAVE_CREATE" and is not required for "FREE_REPLAY".

<goalFile> specifies SignalSet and coverage objects. This file may be omitted, but no coverage can be measured until cover objects are defined. As discussed earlier, other PLI routines are provided to define coverage objects after this point.

<outGoalFile> is optional and is used to specify a file which reached goals will be saved in upon VCS termination. This format is suitable to be used as input <goalFile> or as input to the \$readCoverInfoHV routine.

7.1.2. reportDutStateHV PLI Routine

```
$reportDutStateHV("<outFile>", "<Xmode>");
```

This routine writes the current state of the DUT out to file <outFile>. The file format is intended to be sourced by the hv program; a HvHbv StateVector named "StateVec" is created for the machine named "hvmch_\$design". Each bit of StateVec is assigned the current logic value of the corresponding verilog register. Only registers indicated by the <mapFile> of \$startHV are assigned values. The <Xmode> parameter is a string which indicates what logic value should be used for registers which currently have the logic X value. If <Xmode> is "DC", then those X-ed registers have Don't Care value; if <Xmode> is "CONST", then those X-ed registers will be assigned logic 0 value.

When a register is encountered in the X state, a message is printed (maximum of one message). A comment is written at the end of <outFile> which documents how many registers were in the X state.

7.1.3. informSearchResultsHV PLI Routine

```
$informSearchResultsHV(<#reached_goals>)
```

This PLI routine is used to tell the VCS-resident hv code that the previous search was either successful or not. This information is used to determine applicability of a particular state for use as a future search start point. For example, if a particular state was used as the start state for a formal search, and that search failed after 2 hours, then the state might not be a good point to stop and search from in the future.

The <#reachedGoals> is how many goals were reached by a formal search engine from the state last used by the last call to \$reportDutStateHV.

7.1.4. configStopPointHV PLI Routine

```
$configStopPointH(<trigReg>, <covHandle>, \
    <samplePercent>, <startPercent>, <cycCnt>, <suppCnt>)
```

This PLI routine is used to tell the VCS-resident HV a simulation run should pause and allow control decisions to be considered by the Master process.

<trigReg> is a verilog wire which will be toggled when a “good” stop point has been reached. If <trigReg> is null, then the PLI routine tf_dostop() is called.

<covHandle> specifies the coverage object to which this stop configuration applies. If <covHandle> is -1, then this information applies to all current coverage objects.

<samplePercent> is an integer, in 1/1000 percent units, which specifies an upper bound for how often a state can be “visited” and still be considered “fresh”. For example, if a coverage object has been sampled 1000 times and has been in a given state S 10 of those 1000 times, then S has been visited 10/1000 or 1% of the total sample times. If <samplePercent> is greater than 10, then S is considered fresh and may cause a simulation stop to occur. Note that “state” consists of only the coverage variables used by the coverage object; not all registers of the design.

<startPercent> is an integer, in 1/1000 percent units, which specifies an upper bound for how often a state may be used as a “start state” for formal searches. Each state at the time \$reportDutStateHV is called is considered a “start state” for a formal search.

Note: it is acknowledged that this is not quite always true. However, in the current system, this is very nearly a true statement.

As an example, if a state S has been used for start state 3 times, and 30 formal searches have been performed, then <startPercent> must be greater than 100 (10%) in order for S to be considered “fresh”. A special case happens if a previous search from state S resulted in no goal reached; in this case, S will never be used again as a stop point.

<cycCnt> specifies a maximum number samples which are allowed to pass until a stop is called. This limit is relative to the last goal reached in the coverage object, or relative to the last reset sequence being executed. If <cycCnt> is 0, then a stop will only occur for fresh states.

<suppCnt> is a freshness “suppress” count. Freshness detection is suppressed for this many samples after a new goal is reached.

Note that if either percent is 0, then that metric is turned off for freshness detection. If both percentages are non-0, then a trigger (i.e. stop) condition is the AND of both percentages.

7.2. Verilog TestBench Top Module

The verilog testbench top file is generated by the vera synthesis tool. This “top.v” file instantiates the verilog module being verified, as well as VERA constraint and biasing mechanisms. Refer to appropriate documentation for more details on this process. This section discusses control and interface aspects contain in the top module.

Verilog Control and Status Variables

As outlined earlier in this document, the TCL control code running in the Master process needs to tell the VCS Slave to do a number of things. For example, “execute a reset

sequence”, “replay a generated trace”, “clear dynamically-biased weights”, etc. The mechanisms used to convey these commands are to 1) call various CTG-supplied PLI routines, or 2) set various control flags in the verilog testbench top. These control flags are periodically checked by the VERA VCS code which controls the VCS Slave process. This VERA control code is called “CoverBooster” and is described in a later section of this document.

Flags are defined as verilog “reg”s, and are set to 1 or 0 interactively by using the native VCS command “set”. For example, given a statement:

```
reg foo;
```

Then the following interactive VCS command will set foo to 1.

```
cli_1> set foo=1
```

Control flags which are used in the top module of the verilog testbench are the following:

- hvResetFlag – This flag is used to tell CoverBooster to apply the reset sequence at the next possible time. This calls the reset() task in the design_interface implemented in VERA code. This flag is initialized to 1 so that a reset is performed as the very first action.
- hvReplayTraceFlag – This flag tells CoverBooster to replay a replay trace generated by one of the formal search engines. The filename used to contain the trace is defined historically as “prm_sequence”. This flag is initialized to 0.
- hvClearWeightsFlag – This flag tells CoverBooster to clear dynamically-biased input weights. This calls the reset_weight() task in the dynamic_interface implemented in VERA code. This flag is initialized to 0.
- hvStopFlag – This flag causes CoverBooster to call the VCS \$stop() routine. This flag is initialized to 1, so that CoverBooster stops after applying the first reset sequence; the Master process then calls \$startHV and the actual CTG session can be started.
- hvQuitFlag – This flag causes CoverBooster to exit the main command loop and write out a repeatable test file. VCS then executes the \$stop command and waits for further input. The CTG test generation process may not be continued from this point. This flag is initialized to 0.
- hvBiasWeightsFlag – This flag causes CoverBooster to use information in the next replayed trace to modify primary input weighting information. This flag is initialized to 0.

Vera-Accessible PLI Tasks

The CoverBooster program needs access to many of the PLI routines and verilog flags maintained in the verilog testbench top file. The mechanisms VERA uses for this is via verilog “tasks” which are implemented in top and declared external for CoverBooster in an include file. This section describes those VERA-accessible tasks:

- The startHV task calls the \$startHV PLI routine and is used by CoverBooster when replaying a previously-generated testcase in stand-

alone mode. More detail on this mode is given in a later section of this document. This task prototype is the following:

```
task startHV;
    input rootInstStr;
    input modeStr;
    input mapStr;
    input inFile;
    input outFile;
    reg [799:0] rootInstStr;
    reg [799:0] modeStr;
    reg [799:0] mapStr;
    reg [799:0] inFile;
    reg [799:0] outFile;
```

- The readFlagsHV task is used to read all control flags in the top module. Task prototype is the following:

```
task readFlagsHV;
    output resetFlag;
    output replayTraceFlag;
    output clearWeightsFlag;
    output stopFlag;
    output quitFlag;
    output biasWeightsFlag;
```

- The following tasks clear the indicated flags and take no arguments.

```
task clearClearWeightsFlagHV;
task clearResetFlagHV;
task clearReplayTraceFlagHV;
task clearStopFlagHV;
task clearQuitFlagHV;
task clearBiasWeightsFlagHV;
```

- The following two tasks make direct calls to the VCS routines \$stop and \$finish, respectively.

```
task stopVcsHV;
task finishVcsHV;
```

- The getCoverMetricHV task calls the \$getCoverMetricHV PLI routine. Task prototype is

```
task getCoverMetricHV;
    input covHandle;
    output coverValue;
    integer covHandle;
    integer coverValue;
```

- The printCoverInfoHV task calls the \$printCoverInfoHV PLI routine. If useRchFlag is non-0, then "COMPACT_REACH" is used, else "COMPACT". A value -1 is used for the input coverage handle. Task prototype is:

```
task printCoverInfoHV;
    input fileNameStr;
    input useRchFlag;
    reg [799:0] fileNameStr;
```

- The readCoverInfoHV task calls the \$readCoverInfoHV PLI routine. Task prototype is:

```
task readCoverInfoHV;
    input fileNameStr;
    input useRchFlag;
    reg [799:0] fileNameStr;
```

- The reportDutStateHV task calls the \$reportDutStateHV PLI routine. Task prototype is:

```
task reportDutStateHV;
    input fileStr;
    input modeStr;
    reg [799:0] fileStr;
    reg [799:0] modeStr;
```

7.3. Vera Interfacing - CoverBooster

The CoverBooster program is written in VERA and is designed to interface the VCS Slave process with the rest of the CTG system, and to interact with the design-specific VERA code which biases and constrains the random simulation process. This section outlines the design-independent control flow of CoverBooster. User-specified biasing and constraints, compilation methodology, etc are beyond the scope of this document.

Two aspects of CoverBooster are described here: the top-level control flow, and stimulus/sampling top-level clocking strategy.

7.3.1. CoverBooster Clocking Strategy

Clocking strategy in the CoverBooster context deals with when inputs are applied and coverage measured with respect to the SystemClock used with VERA. This is a separate issue from “clocking strategy” for the Design Under Test itself. The DUT clocking strategy might deal with clock dividers, positive/negative registers, edge detectors, etc.

It is expected that part of environmental specification is to synthesize the DUT clock generator in terms of a single SystemClock that CoverBooster knows about.

Given a SystemClock with period T, Figure 4 illustrates when inputs are applied to the DUT and when coverage is sampled.

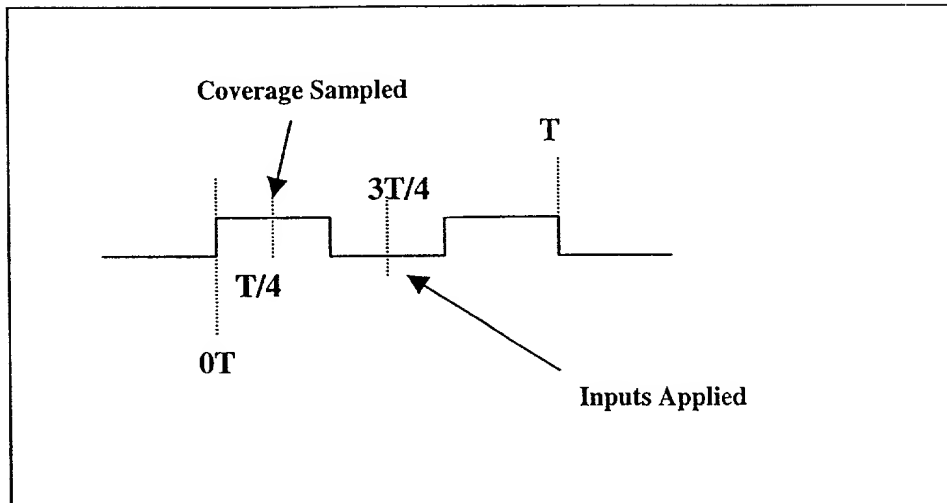


Figure 4. CoverBooster Clocking Strategy

As shown in Figure 4, all inputs are applied $\frac{1}{4}$ clock period before the positive clock edge. Coverage is measured $\frac{1}{4}$ cycle after the positive clock edge. Formal search engines are used $\frac{1}{4}$ cycle before the positive clock edge.

Note that sample points for individual coverage objects may be relative to different clocks in DUT (e.g. see the discussion for the `$defineFsmCoverHV()` PLI routine). Coverage is measured in CoverBooster to determine when all goals are reached, and to determine if a replayed trace was successful or not.

7.3.2. CoverBooster Control Flow

This section describes key control flow functions in the CoverBooster program.

7.3.2.1. Main CoverBooster Program

Top level control flow is illustrated by the following pseudocode.

```

Program Begin
  initialization and read plus args
  wait for first posedge clock
  wait for next negedge clock
  wait for  $\frac{1}{4}$  more of the clock cycle
  if (testcase replay mode)
    testcaseReplay();
    $finish()
  Endif
  ForEver ()
    ReadControlFlags()
    If (resetFlag)
      Call interface reset() task
      ClearResetFlag()
    ElseIf (biasWeightsFlag)
      Remember biasing dynamic request

```

```

        ClearBiasWeightsFlag()
    Elseif (clearWeightsFlag)
        Call resetWeights() in dynamic interface
        ClearClearWeightsFlag()
    Elseif (replayTraceFlag)
        replaySequence()
        Report HIT/MISS to Master
        Setup for dynamic combinational biasing
        ClearReplayTraceFlag()
    Elseif (stopFlag)
        $stop()
        ClearStopFlag()
    Elseif (quitFlag)
        ClearQuitFlag()
        Break
    Else
        If (dynamic biasing)
            DynamicBiasedDrive()
        Else
            UserBiasedDrive()
        Endif
        Update coverage metric
        Increment clock counter
        Wait for negedge clock
        Wait for ¼ clock period
    Endif
    If (no goals left)
        Break
    Endif
EndForEver
If (generating test requested)
    Finish writing replay-able test
Endif
$stop()
Program End

```

7.3.2.2. replaySequence VERA Function

This function is called to replay a trace (sequence of inputs to apply over one or more clock cycles). This trace was generated by one of the formal search engines, and should result (usually) in a new goal being hit.

The file which contains the trace information contains all clocking and input assignment information. The replaySequence function simply reads the file and applies logic 1/0 to inputs and waits for clocks as instructed. File format is one token per line, and is the following:

- // - Comment lines begin with two forward slashes
- !begin! – This is the first token of the replay trace
- <portName> - This token is a primary input port to the Design Under Test. It is followed in the file by
- <logicValue> - This token is a '0' or '1' and is the logic value to drive on the port indicated by the previous token.

- **!clock!** – A clock token specifies that a single clock cycle is applied to the design. I.E. one clock period of simulation time advances.
- **!end!** – This token specifies the end of the replay trace.

If requested, dynamic biasing weights are accumulated as the input ports are driven; this information is used later for combinational dynamic biasing.

The following pseudocode illustrates control flow for this function:

```
Function Begin
    OldCover = getCoverageNumber()
    Open trace file
    Text = getNextLineFromFile();
    If (Text != "!begin!")
        Error
    Endif
    ForEver()
        Text = getNextLineFromFile();
        If (Text == "!end!")
            If (biasingWeights)
                accumulateWeights();
            Endif
            Wait for negedge clock
            Wait for ¼ clock period
            Break
        Endif
        If (Text == "!clock!")
            If (biasingWeights)
                accumulateWeights();
            Endif
            Wait for negedge clock
            Wait for ¼ clock period
            Continue
        Endif
        LogicValue = getNextLineFromFile();
        DriveDataOnPort(Text, LogicValue)
    EndForEver

    NewCover = getCoverageNumber()
    If (OldCover > NewCover)
        Report HIT
    Else
        Report MISS
    Endif
Function End
```

7.3.2.3. testcaseReplay VERA Function

This function is called when a complete testcase is to be replayed in stand-alone mode. The test file format is an extension of the trace file format. The test file format, in addition to the tokens described earlier for trace file, is the following:

- **!test_begin!** – This token indicates the begin of a testcase.
- **!run!** – a VCS run should be started. The next line in the file is:

- **<cycleCount>** - specifies the number of clock cycles that should be applied to the design.
- **!reset!** – a reset sequence is to be applied
- **!bias_weights!** – weights should be accumulated when the next trace is replayed.
- **!replay!** – a trace is to be replayed. This token is followed by the token sequence described earlier for trace replay file format.
- **!clear!** – dynamic bias weights should be cleared
- **!test_end!** – This token signifies the end of the testcase

Pseudocode for the testcaseReplay function is the following:

```

Function Begin
    Apply the first reset sequence
    ForEver()
        opCode = getNextLineFromFile();
        If (opCode == "!test_begin!")
            ReadCoverInfo() // goal definitions
        ElseIf (opCode == "!reset!")
            Call reset() function
        ElseIf (opCode == "!run!")
            ClkCnt = getNextLineFromFile();
            For (i=0;i<clkCnt;i++)
                If (dynamic comb biasing)
                    Dynamic_biased_drive()
                Else
                    Random_drive()
            Endif
            Wait for negedge clock
            Wait for ¼ clock period
        EndFor
        ElseIf (opCode == "!replay!")
            ReplaySequence()
        ElseIf (opCode == "!clear!")
            Reset_weights()
        ElseIf (opCode == "!bias_weights!")
            Next replay biases weights
        ElseIf (opCode == "!test_end!")
            break
        Else
            Error
        EndIf
    EndForEver
Function End

```

7.4. Verilog Program Control – Usage Scenarios

There are at least three usage scenarios which the verilog-resident CTG code must support. Verilog “plus args” and PLI routines are used to communicate which mode analysis is intended.

Note: plus args are user-defined verilog invocation switches which are accessible from either PLI routines or from VERA code. Plus args which are defined for use by CTG are the following:

- `+ctg_mode=<mode_str>` - This plus arg is used to specify which mode VCS is being used for. Currently recognized values are "SLAVE_CREATE", "FREE_REPLAY", and "FREE_CREATE". These are described later in this section.
- `+root_dut=<dut_module>` - This plus arg specifies which module in verilog is the root for formal search methods.
- `+ctg_testfile=<testfile>` - In replay mode, this plus arg specifies a testcase file which contains control flow for replay of a previously-created testcase. In test generation mode, this file is written with the generated test.
- `+ctg_ingoals=<input_goalfile>` - This plus arg specifies an input file which defines coverage goals for the analysis session. This file optionally also contains goals which were covered in previous tests, and are therefore not considered in the current session.
- `+ctg_outgoals=<output_goalfile>` - This plus arg specifies a file to which reached goal information will be written upon process termination. This file format is suitable for use as `<input_goalfile>` in subsequent sessions.

The remainder of this section discusses key usage scenarios.

7.4.1. Testcase Creation

This is the classic CTG mode where formal search and explicit state simulation are interleaved to maximize coverage of user-defined goals. Assuming the previously-discussed `hv_start_vcs_session` TCL procedure is used, invocation of the compiled VCS simulator looks like the following:

```
vcs.exe -s -l vcs.log +ctg_testfile=TestOut \  
+ctg_mode=SLAVE_CREATE
```

This invocation may be slightly different depending on TCL knob settings, and argument used to the `hv_start_vcs_session` procedure.

The result of this command will be to write a testcase control file "TestOut.ctg", as well as reached goals file "TestOut.vgol" when VCS terminates.

7.4.2. Testcase Replay

Given a testcase created in a previous CTG session, that test may be replayed in stand-alone mode using the following simulator invocation.

```
vcs.exe -s -l vcs.log +ctg_testfile=<testfile> \  
+ctg_mode=FREE_REPLAY +root_dut=<dut_module>
```

Where `<testfile>` was the testcase filename used in testcase creation. `<dut_module>` is the verilog root module name for formal search engines during the creation session.

7.4.3. Leveraging User-Written Tests

A common usage for CTG is expected to be that of augmenting hand-written test coverage by use of CTG mechanisms. In this scenario, the hand-written test will cover some goals, and CTG will be used to classify those goals not reached by the hand-written tests. This section describes how this can happen

To collect goals which are reached by the hand-written tests, the following actions are carried out:

1. The user, presumably with some UI help from CTG, defines which goals are of interest; these goals are written to a file in a format readable by the \$startHV and \$readCoverInfoHV PLI routine.
2. The customer links the CTG PLI library to their VCS simulation.
3. Upon invocation, the user must call the \$startHV PLI routine to initiate a CTG collection session. This may either be done using the “-f” verilog switch, or by using VCS interactive mode. The <root_inst> parameter is required to be that which will later be formally searched. <mode> is “FREE_CREATE”. <regMapFile> is “”. <inGoalFile> is the goalfile created in step 1). <outGoalfile> is specified; <outGoalFile>.ctg and <outGoalFile>.vgol will be written as previously described.
4. If the user has previously reached goals from an earlier session, they may be removed from current consideration by calling the \$readCoverInfoHV PLI routine.

When this session ends, <outGoalFile>.vgol will contain the goals which were reached in this session, and is suitable as input to a later CTG session.

5

Appendix 2

VERA VERIFICATION SYSTEM USER'S MANUAL

Synopsys, Inc.

10

Vera[™] Verification System

User's Manual

Version 4.0

Synopsys Inc.
1860 Embarcadero Rd., Suite 260
Palo Alto, California 94303
Tel (415) 812-1800 • Fax (415) 812-1820
<http://www.synopsys.com> • vera-info@synopsys.com

Vera, Vera-VS, Vera-SV, Vera-VL, Vera-HVL, Vera Verification System, Verity, Verity ToolKit, ISDB, ISDB-cycle, and PowerFault are trademarks of Synopsys Inc. Magellan, PowerSim, and SimWave are registered trademarks of Synopsys Inc. All other trademarks are the property of their respective owners.

Copyright © 1996, 1997, 1998, 1999 by Synopsys, Inc.
All rights reserved.

PATENTS PENDING.

Vera -Doc.4.0 (1999)
Revision 1
Last updated: January 31, 1999

Doc.4.0 (1999)

Table of Contents

Preface, xi

1. Introduction to Vera, 17

- 1.1 High Level Verification, 17
- 1.2 Why Vera?, 18
- 1.3 What Can Vera Do?, 18
- 1.4 Overview, 20
 - 1.4.1 How Vera Is Used, 21
 - 1.4.2 Vera Components, 21
 - 1.4.3 Vera/HDL Flow of Data, 22
- 1.5 A Small Example, 23
- 1.6 Multiple-Clock / Multiple-Module Systems, 27
- 1.7 Simple Example of a 2-Clock System, 28

2. Installation and Setup, 31

- 2.1 Installation, 31
- 2.2 Licensing, 31
 - 2.2.1 Licensing for Current FlexLM Users, 31
 - 2.2.2 Licensing for Other Users, 31
 - 2.2.3 Controlling Vera License Queueing, 34
- 2.3 Running Vera with Chronologic VCS, 34
- 2.4 Running Vera with Verilog-XL, 35
- 2.5 Running Vera with the Verilog ModelTech Simulator, 37
- 2.6 Running the Vera Stand-Alone Simulator, 38
- 2.7 Testing the Installation, 39
- 2.8 Vera Support, 39

3. Vera-HVL: The Language, 41

- 3.1 Lexical Conventions, 41
 - 3.1.1 White Space, 41
 - 3.1.2 Comments, 41
 - 3.1.3 Statement Blocks, 42
 - 3.1.4 Identifiers and Keywords, 42
 - 3.1.5 Strings, 44

3.1.6	Numbers, 44
3.2	Data Types and Variable Declaration, 45
3.2.1	Integers, 45
3.2.2	Bits, 46
3.2.3	Strings, 46
3.2.4	Bind_vars, 46
3.2.5	Events, 47
3.2.6	Enumerated Types, 47
3.2.7	cast_assign(), 47
3.3	Arrays, 48
3.3.1	Arrays, 48
3.3.2	Associative Arrays, 49
3.4	Strings, 50
3.4.1	Printing Strings, 50
3.4.2	Valid Operators, 51
3.4.3	String Class Methods, 51
3.4.4	String Class Methods for Matching Patterns, 53
3.4.5	String Class Methods for Type Conversion, 56
3.4.6	Additional System Functions, 58
3.5	Enumerated Types, 59
3.5.1	Enumerated Types in Numerical Expressions, 60
3.5.2	Increment and Decrement Operations on Enumerated Types, 60
3.6	Vera Operators, 61
3.6.1	Operators, 61
3.6.2	Concatenation, 63
3.7	Variable Assignment, 63
4.	Basic Vera Programming, 65
4.1	Vera Programming Overview, 65
4.2	Program Structure, 66
4.3	Signal Declarations, 67
4.3.1	Vera-HDL Interface Specifications, 67
4.3.2	Virtual Ports, 70
4.3.3	Bind, 70
4.3.4	Using Ports and Binds, 72
4.3.5	Dynamic Binding, 75
4.4	Signal Operation, 80
4.4.1	Synchronization, 81

- 4.4.2 Drive, 81
- 4.4.3 Sample, 84
- 4.4.4 Expect, 84
- 4.4.5 Value Change Alert (VCA), 87
- 4.4.6 Implicit Synchronization, 89
- 4.5 Asynchronous Operations, 89
 - 4.5.1 async, 90
 - 4.5.2 Sub-Cycle Delays, 90
- 4.6 Subroutines, 91
 - 4.6.1 Functions, 91
 - 4.6.2 Tasks, 92
 - 4.6.3 Return, 93
 - 4.6.4 Breakpoint, 94
 - 4.6.5 Static Variables, 94
 - 4.6.6 Subroutine Arguments, 94
 - 4.6.7 External Definitions, 96

Example 1: Vera Basics, 99

5. Sequential Control, 105

- 5.1 If-else Statements, 105
- 5.2 Case Statements, 106
- 5.3 Randcase Statements, 107
- 5.4 Repeat Loops, 107
- 5.5 For Loops, 108
- 5.6 While Loops, 109
- 5.7 Break and Continue, 109
 - 5.7.1 Break, 109
 - 5.7.2 Continue, 110

6. Concurrent Control, 111

- 6.1 Fork and Join, 111
 - 6.1.1 Fork and Join Control, 113
 - 6.1.2 Shadow Variables, 116
- 6.2 Events, 116
 - 6.2.1 Triggers, 117
 - 6.2.2 Sync, 118
 - 6.2.3 Event Variables, 119

- 6.3 Semaphores, 122
 - 6.3.1 Conceptual Overview, 122
 - 6.3.2 alloc(), 122
 - 6.3.3 semaphore_get(), 122
 - 6.3.4 semaphore_put(), 123
 - 6.3.5 Semaphore Example, 123
- 6.4 Regions, 125
 - 6.4.1 Conceptual Overview, 125
 - 6.4.2 alloc(), 125
 - 6.4.3 region_enter(), 125
 - 6.4.4 region_exit(), 126
 - 6.4.5 Region Example, 126
- 6.5 Mailboxes, 127
 - 6.5.1 Conceptual Overview, 127
 - 6.5.2 alloc(), 128
 - 6.5.3 mailbox_put(), 128
 - 6.5.4 mailbox_get(), 128
 - 6.5.5 Mailbox Example, 129
- 6.6 Timeout Limit, 130
- 6.7 Backward Compatibility, 131
 - 6.7.1 mailbox_send(), 131
 - 6.7.2 mailbox_receive(), 131

Example 2: Flow Control, 133

7. Predefined Vera Tasks, 141

- 7.1 Vera I/O, 141
 - 7.1.1 Output, 141
 - 7.1.2 File IO, 142
- 7.2 Random Number Generators, 145
- 7.3 Simulation Errors, 146
 - 7.3.1 Error Handling, 146
 - 7.3.2 Debug Support Routines, 148
- 7.4 Simulation Control, 149
- 7.5 Reading Plus Arguments, 150

8. Object-Oriented Programming, 153

- 8.1 Classes and Objects, 154

- 8.2 Objects and Instance of Classes, 155
- 8.3 Accessing Object Properties, 156
- 8.4 Using Object Methods, 156
- 8.5 Constructors, 156
- 8.6 Class Properties, 157
- 8.7 this, 157
- 8.8 Assignment, Re-naming and Copying, 158
- 8.9 Subclasses and Inheritance, 159
- 8.10 Overridden Members, 160
- 8.11 super Class, 161
- 8.12 Casting, 161
- 8.13 Chaining Constructors, 161
- 8.14 Data Hiding and Encapsulation, 162
- 8.15 Philosophy, 163
- 8.16 Abstract Classes and Virtual Methods, 163
- 8.17 Finding the Right Method, 164
- 8.18 Polymorphism: Dynamic Method Lookup, 166
- 8.19 Out of Block Declarations, 167
- 8.20 External Classes, 167
- 8.21 Typedef, 168

Example 3: Object-Oriented Programming, 169

9. Automated Stimulus Generation, 177

- 9.1 Introduction to Stimulus Generation, 177
- 9.2 Stimulus Generation Overview, 178
- 9.3 Random Packet Generation, 178
 - 9.3.1 Random Variables, 178
 - 9.3.2 randomize(), 180
 - 9.3.3 Constraint Blocks, 186
 - 9.3.4 Boundary Conditions, 194
- 9.4 Data Packing and Unpacking, 196
 - 9.4.1 Property Attributes, 196
 - 9.4.2 Packing Methods, 197
 - 9.4.3 Unpacking Methods, 198
 - 9.4.4 Pack and Unpack Example, 198

9.4.5 Details of Pack and Unpack, 199

Example 4: Stimulus Generation, 201

10. Vera Stream Generator, 207

10.1 VSG Overview, 207

10.2 Production Definitions, 207

10.2.1 Production Items, 208

10.2.2 Weights, 209

10.3 Production Controls, 210

10.3.1 If-else Statements, 210

10.3.2 Case Statements, 210

10.3.3 Repeat Loops, 211

10.3.4 Break and Continue, 211

10.4 Value Passing, 212

10.4.1 Value Declaration, 212

10.4.2 Value Passing Functions, 213

Example 5: Vera Stream Generator, 217

11. Functional Coverage, 223

11.1 Coverage Overview, 223

11.2 Coverage Definitions, 224

11.2.1 Expressions within Coverage Definitions, 225

11.2.2 Coverage Declarations, 225

11.2.3 Measure of Coverage, 232

11.2.4 Coverage Goal, 233

11.2.5 Coverage Options, 233

11.2.6 Initialization, 234

11.3 Instantiation and Triggering, 235

11.4 Coverage Feedback: The Query Function, 237

11.5 Coverage Administration, 238

11.6 Cross Coverage, 240

11.7 Backward Compatibility, 244

11.7.1 Coverage Options, 244

11.7.2 Coverage Reporting, 245

11.7.3 Conditional Coverage, 245

Example 6: Coverage Analysis, 247

12. Multiple Module Support, 253

- 12.1 Introduction, 253
- 12.2 Overview, 254
- 12.3 Project Files, 255
- 12.4 Configuration Files, 255
 - 12.4.1 Timescale Statement, 255
 - 12.4.2 Clock Statement, 256
 - 12.4.3 Clock_alias Statement, 256
 - 12.4.4 Connect Statement, 256
 - 12.4.5 Veritask Statement, 257
- 12.5 Generating a Project-based Configuration, 257

Example 7: Multiple Modules, 259

13. Vera-CORE, 265

- 13.1 Introduction, 265
- 13.2 Vera-CORE Usage Model For IP Vendors, 266
- 13.3 Vera-CORE Usage Model for IP Users, 267

Example 8: Vera-CORE, 269

14. Vera-to-HDL Task Calls, 275

- 14.1 Calling HDL Tasks from Vera, 275
 - 14.1.1 HDL Tasks in Vera: Outputs/Inouts, 276
 - 14.1.2 Type Checking HDL Tasks in Vera, 276
- 14.2 Calling Vera Tasks from Verilog, 277
 - 14.2.1 Declaring Vera Tasks for Export, 277
 - 14.2.2 Calling Exported Vera Tasks, 277
 - 14.2.3 Exporting External Tasks, 278

15. Calling C/C++ Functions, 279

- 15.1 Declaration and Invocation, 279
- 15.2 UDF Arguments, 279
 - 15.2.1 Vera UDF Arguments, 279
 - 15.2.2 C/C++ UDF Arguments, 280

- 15.3 Linking UDFs to Vera, 281
 - 15.3.1 Vera UDF Table, 281
 - 15.3.2 Object Files and vera_local.dll for Verilog, 282
 - 15.3.3 Object Files and vera_mti.dll for VHDL, 283
- 15.4 PLI Support Procedures, 284
- 15.5 Handling Special Events, 285

Example 9: User-Defined Functions, 287

16. VERA-SV API, 293

- 16.1 Vera-SV Motivations, 293
 - 16.1.1 Hardware/Software Co-Verification, 293
 - 16.1.2 Distributed Simulation, 294
 - 16.1.3 Master-Slave Configuration, 294
- 16.2 Vera-SV Overview, 294
- 16.3 The Vera Side, 295
 - 16.3.1 Initializing Connections, 295
 - 16.3.2 Servicing Remote Calls, 297
 - 16.3.3 Submitting Remote Calls, 297
 - 16.3.4 Managing errors, 298
 - 16.3.5 Master-Slave Configuration, 299
 - 16.3.6 Separate Client and Server Example, 300
 - 16.3.7 Combination Client-Servers Example, 301
- 16.4 The C/C++ Side, 303
 - 16.4.1 Calling a Vera Task from a C-program Example, 304
 - 16.4.2 Calling a C Function from a Vera Program Example, 306
- 16.5 Troubleshooting, 307
- 16.6 Backward Compatibility, 308

17. Testbench Setup and Usage Notes, 309

- 17.1 Creating Testbenches for Verilog Designs, 309
 - 17.1.1 Vera Template Generator, 309
 - 17.1.2 Connecting Vera Testbenches to Verilog Designs, 312
 - 17.1.3 Multiple Clocking Domains, 319
- 17.2 Creating Testbenches for VHDL Designs, 319
 - 17.2.1 Files for Test-top Connections, 319
 - 17.2.2 Connecting Vera Testbenches to VHDL Designs, 325
 - 17.2.3 Directly Connecting Testbenches to the DUT, 330

- 17.2.4 Multiple Clocking Domains, 332
- 17.3 VHDL Testbench Usage Notes, 333
 - 17.3.1 Valid VHDL Signal Types, 333
 - 17.3.2 VHDL Plus Arguments, 333
 - 17.3.3 Features Not Yet Supported for VHDL, 333

18. Compilation, 335

- 18.1 Compiler Overview, 335
- 18.2 Preprocessor, 336
 - 18.2.1 File Inclusion, 336
 - 18.2.2 Text Macros, 337
 - 18.2.3 Conditional Compilation, 337
- 18.3 General Options, 338
- 18.4 Compile Options, 340
- 18.5 Running Vera with HDLs, 343
- 18.6 Modular Compilation, 346
- 18.7 Dynamic Loading of Vera Object Modules, 348
- 18.8 Running Vera Stand-alone, 349

19. Vera Source-Level Debugger, 353

- 19.1 Overview, 353
- 19.2 Interactive Debugging, 353
- 19.3 Contexts, 354
- 19.4 Debugger Expressions, 354
- 19.5 Text-Based Debugger Commands, 354
 - 19.5.1 Execution Control, 356
 - 19.5.2 Displaying and Updating Data, 357
 - 19.5.3 Accessing Context Information, 358
 - 19.5.4 Accessing Source Files, 359
 - 19.5.5 Watch Commands, 360
 - 19.5.6 Debugging Environment Control, 362
 - 19.5.7 Miscellaneous Commands, 362
- 19.6 Graphical Debugger, 363
 - 19.6.1 Usage Notes, 363
 - 19.6.2 Notes for Tcl/Tk Users, 364

Appendix A: Quick Reference, A1

- A.1 Vera Files, A1
- A.2 Vera System Functions and Tasks, A1
- A.3 Compiler Switches, A4
- A.4 Vera Debugger Commands, A6

Appendix B: Vera 4.0 BNF Diagrams, B1

- B1. Vera-HVL: The Language, B1
 - B1.1 Program Structure, B1
 - B1.2 Interface Definition, B3
 - B1.3 Variable Definition, B5
 - B1.4 Expressions, B6
 - B1.5 Statements, B8
 - B1.6 Interface Signal Control, B9
 - B1.7 Sequential Control, B12
 - B1.8 User Defined Types, B14
 - B1.9 Functions, B15
 - B1.10 Tasks, B17
 - B1.12 Classes, B19
 - B1.13 Coverage Blocks, B21
 - B1.14 Coverage Objects Definition, B22
 - B1.15 Vera Stream Generator, B26
 - B1.16 HDL Tasks, B28
 - B1.17 UDF Declarations, B28
- B2. Vera System Tasks & Functions, B29
 - B2.1 Concurrent Control, B29
 - B2.2 String operations, B31
 - B2.3 Input/Output, B33
 - B2.4 Coverage Control, B35
 - B2.5 VSG Value Passing Support, B36
 - B2.6 Debug Support, B37
 - B2.7 Simulation Timing, B39
 - B2.8 Signal Control, B39
 - B2.9 Automated Stimulus Generation, B40
 - B2.10 Data compaction, B40
 - B2.11 Polymorphism support, B41
 - B2.12 Vera-SV API, B42
- B3. Vera Simulator, B44

Appendix C: Perl Expressions, C1

About This Book

The *Vera 4.0 User's Manual* is designed to introduce you to Vera-HVL and explain how to develop Vera testbenches for use with Verilog and VHDL designs. The book discusses the basic components of the Vera language and progresses to advanced testbench design topics. The book is designed to serve as a supplemental teaching guide as well as a reference guide for the advanced user.

Audience

This book is intended to be used by engineers with experience with either Verilog or VHDL. Because Vera is used to create testbenches that link with Verilog and VHDL hardware designs, you should have a fundamental understanding of the description language you are using. This book also assumes a limited verification background. Finally, this book assumes use of a high-level programming language such as C, C++, or Java.

How to Use This Book

The first chapter of this book is designed to give an overall view of Vera-HVL and explain why it is different than other industry standards. Chapters 2-7 describe some of the basic Vera components. Advanced topics are covered in chapters 8-19.

These are the chapters and a brief description of each:

- **Chapter 1. - Introduction to Vera**

This chapter introduces Vera and briefly describes some of the major features that make it one of the premier verification tools in industry. This chapter discusses the flow of data using Vera and briefly explains how this flow relates to testbench design and usage. Finally, this chapter includes brief examples that illustrate basic Vera usage models.

- **Chapter 2. - Installation and Setup**

This chapter details how to install Vera and setup a user environment. The chapter includes licensing information. It also includes instructions on how to recompile HDL simulators to link to the Vera simulator. Support information is also located in this chapter.

- **Chapter 3. - Vera-HVL: The Language**

This chapter introduces the basic lexical conventions of Vera-HVL. It discusses the various data types and variable types. This chapter also details some of the more complex data types, including arrays and enumerated types.

- **Chapter 4. - Basic Vera Programming**

This chapter discusses the basic components of a Vera program. It includes discussions of signal declarations and signal operations, including driving and sampling signals. This chapter also includes a detailed discussion of creating subroutines within Vera.

- **Chapter 5. - Sequential Control**

This chapter covers the sequential flow constructs of Vera, including if statements, case statements, and for and while loops. It also includes information on sequential flow control.

- **Chapter 6. - Concurrent Control**

This chapter discusses the concurrent flow control capabilities of Vera. It includes a detailed discussion of concurrent processes. It also covers triggers, semaphores, regions, and mailboxes, which are key components to controlling concurrent processes.

- **Chapter 7. - Predefined Vera Tasks**

This chapter documents the majority of Vera's predefined tasks and functions. Included in this chapter are the tasks and functions associated with file input and output, randomization, and error handling.

- **Chapter 8. - Object-Oriented Programming**

This chapter is a brief tutorial on object oriented programming. It describes the basic components of object oriented programming and details some of the more advanced features such as inheritance and polymorphism.

- **Chapter 9. - Automated Stimulus Generation**

This chapter is a detailed discussion of Vera's stimulus generation features. This chapter covers the randomization of test packets, constraint-driven stimulus generation, and data packing and unpacking.

- **Chapter 10. - Vera Stream Generator**

This chapter discusses the Vera Stream Generator. It includes an introduction to sequence generation and includes complete coverage of the system tasks and functions incorporated into the stream generator that allow data passing between productions.

- **Chapter 11. - Functional Coverage**

This chapter begins with a brief overview of functional coverage and explains the inherent advantages over code coverage. It discusses the use of monitor bins to track simulation activities. This chapter also details how to set up complex coverage objects that can be used for cross-correlation between events.

- **Chapter 12. - Multiple Module Support**

This chapter documents Vera's multiple module support. It details how to create testbenches using multiple main programs and incorporate them into a single testbench. It covers generation of testbenches using this methodology as well as specific enhancements designed to ease testbench creation.

- **Chapter 13. - Vera-CORE**

This chapter introduces Vera-CORE, a feature designed to assist IP vendors and their clients. This chapter includes usage models for both the IP vendors and the core users.

- **Chapter 14. - Vera-to-HDL Task Calls**

This chapter provides information on how to make Vera task calls from HDLs and vice versa. The specific constructs are discussed along with some of the inherent limitations.

- **Chapter 15. - Calling C/C++ Functions**

This chapter discusses how to incorporate user-defined functions written in C or C++ with your Vera testbench. It includes information on how to link in dynamic libraries as well as how to call those functions from Vera.

- **Chapter 16. - VERA-SV API**

This chapter covers Vera-SV, which allows you to run distributed simulations with some Vera modules and some C modules. It includes information on system tasks and functions designed to enhance the distributed simulation capabilities of Vera.

- **Chapter 17. - Testbench Setup and Usage Notes**

This chapter discusses how to connect testbenches to the design under test. It includes instructions on how to connect HDL signals to the Vera testbench. It also details how to set up multiple clocking domains in both Verilog and VHDL.

- Chapter 18. - Compilation

This chapter covers all of the compiler options used when compiling Vera source code. It also includes details on how to use the Vera simulator stand-alone. This chapter discusses modular compilation as well.

- Chapter 19. - Vera Source-Level Debugger

This chapter describes the Vera source-level debugger. It describes how both the text-based and graphical versions of the debugger are used. It explains each of the major debugging features incorporated into the debugger.

Copyright © 2000 Synopsys, Inc.

Conventions

This book uses several style and syntax conventions:

- Vera keywords are printed in **bold**.
- Variables are printed in *italics*.
- Signal names, bind names, and user-defined functions are printed as: `signal name, bind name, user function`.
- Vera statements and variable definitions are printed as:

`command_keyword (variable, argument);`

variable - The definition of *variable* is described here.

argument - The definition of *argument* is described here.

- Optional arguments are encapsulated by bold brackets (**[]**):

`command_keyword (variable [[], argument]);`

Copyright © 2000 Synopsys, Inc.

1. Introduction to Vera

This chapter discusses the current industry standards for high-level design verification, and it briefly details the history of Vera. It includes these sections:

- High Level Verification
- Why Vera?
- What Can Vera Do?
- Overview
- A Small Example
- Multiple-Clock / Multiple-Module Systems
- Simple Example of a 2-Clock System

1.1 High Level Verification

Technology is advancing today perhaps faster than ever. New innovations in both hardware and software are extending the capabilities of our computers and building a foundation for an even more accelerated world of changes and technological advancements. The speed at which these changes are taking place demands that industry engineers be able to design, build, and debug new products with increasing accuracy while still producing them fast enough to meet industry standards.

The stress on design integrity is no more evident than in the design and manufacture of integrated circuits. And while engineers have managed to keep designs ahead of the technology curve thus far, the complexity of current designs and the time to verify their functions is becoming a serious concern for chip designers.

To help in the design process, a number of programming languages have been developed to simulate and test hardware design. To that end, Verilog and VHDL have grown increasingly popular in testing design specifications. The power they bring to the engineer to ensure design integrity has paved the way for still more sophisticated designs.

However, in this design world, the reality is that upwards around 75% of the time spent to get a chip from inception to market is spent verifying chip design. Even with the power that Verilog and VHDL bring, designs are becoming increasingly difficult and time consuming to validate sufficiently. To continue to create increasingly complex systems on a chip at the pace demanded by the public, engineers need more powerful tools. Hence, Vera was born.

1.2 Why Vera?

The key to understanding the difference between the current HDLs and Vera-HVL lies in the inception of each. The HDLs are designed to be Hardware Description Languages. As such, they are adequate for developing hardware models. Vera-HVL is designed to act as a Hardware *Verification* Language. Its role exists in an entirely different scope: the verification of complex ICs and systems.

The current HDLs are adept at creating hardware models, which are made up primarily of static concurrent processes that simulate individual parts. However, the HDLs provide a level of abstraction that is far too narrow to develop useful test environments. The lack of concurrent flow control devices greatly limits the use of multiple threads to simulate a real test environment. Engineers have resorted to programming language interfaces that provide links to C, C++, and Perl, but even these fall short because these other languages have no concept of hardware, timing, or concurrency.

Vera-HVL provides the necessary abstraction level to develop reliable test environments. Through the use of complex synchronization and timing mechanisms, concurrent processes can be run that simulate a dynamic test environment. The implementation of the object oriented programming methodology further enhances the ability of the language to work with complicated designs and sophisticated test benches.

Add to this Vera's self-checking capabilities, automated stimulus generation, dynamic coverage analysis, and modular approach to programming, and the power of Vera is quite apparent.

1.3 What Can Vera Do?

Vera is designed to be a hardware verification language. To accurately simulate complex test benches that interact with complicated designs, Vera has a host of features available to its users.

Automated Stimulus Generation

As chips have grown increasingly complex, the ability to write and perform tests on all the functions of a system has become nearly impossible. Testing for specific behavior, while possible, is far too time-consuming to be a viable verification method. Instead, engineers need a method of testing the device under varying conditions and stimuli. To meet these needs, Vera provides the engineer with automated stimulus generation.

The automated stimulus generation process allows engineers to use random stimuli to drive the device under test. By randomizing the stimuli, engineers can simulate the device under more conditions and in less time than with manually-written code.

Vera further enhances the automated stimulus generation feature with constraints that give the engineer control over the random testing of the chip. The constraint mechanism grants the engineer two key advantages: (1) the engineer can weight the tests so that certain sets of stimuli occur more often to simulate a more real environment, and (2) the engineer can use the constraints such that the current model state affects future tests.

Coverage Analysis

Because of the complexity of chips and broad scope of functions that need to be tested for any given chip or system, the completeness of verification is a key issue. While code coverage yields information about specific lines of code, it does not give any detailed information concerning functionality. Thus, code coverage is no longer sufficient for complex model verification.

Vera supports a powerful coverage analysis suite that assures the engineer of the desired coverage. Specific activities are monitored to ensure that all legal states and transitions have been tested. These activities can also be time-stamped to check that specific events happen simultaneously.

Vera provides two methods of coverage analysis: open-loop and closed-loop analysis. Open-loop analysis performs the tests and then creates a report summarizing the results. Closed-loop analysis offers significantly greater power. Using closed-loop analysis, the test suite identifies areas of sparse coverage and uses that information to drive subsequent tests. This approach guarantees completeness of verification while reducing effort and time spent.

Virtual Ports

Vera provides a simple way to re-group or re-name interface signals by creating virtual ports. Virtual ports allow the engineer to group interface signals so that they can be used more easily. These are particularly useful when calling subroutines that you want to act on various interface signals. By grouping the signals, you can pass them to the subroutine as an argument, allowing you to create signal-independent subroutines. The logical grouping reduces the amount of code required to run tests, and makes it easier to comprehend.

Distributed Simulation

Distributed simulation is often used with, but not limited to, hardware/software co-verification. It involves verifying large designs by partitioning verification tasks to run concurrently across multiple processors. Vera supports distributed simulation by enabling multiple Vera programs and C/C++ applications to execute remote calls to each other across a computer network. This is particularly useful when verifying complicated systems that tax computer resources. Distributing the simulation across multiple servers makes the simulation faster and more efficient.

Object Oriented Programming

Object-oriented programming is at the very core of Vera. The implementation is clear, straightforward and elegant, assimilating and extending the best of the features found in C++ and Java. By using the object-oriented methodology, programs are easier to design, easier to develop, easier to debug, easier to maintain, easier to share, and easier to re-use.

Object-oriented programming leads naturally to grouping related code and data together, keeping these sections small, easy to understand, easy to debug and easy to maintain. The result is a discipline and program structure that is useful for small programs, but has a dramatic, powerful impact on your productivity as soon as your program becomes large or in any way complex.

Verification-Specific Advantages

Several other key features have been implemented in Vera to enhance its verification capabilities:

- Vera supports project-based configuration. Project-based configuration allows the engineer to create test benches that are independent of the actual connections to the simulation environment, and connect those testbenches to any point in the simulation hierarchy without the need for a test-top configuration. So, the engineer can run separate test benches with separate program modules concurrently.
- Vera provides a set of complex synchronization mechanisms that allow you to use concurrent processes in a much more dynamic way. The use of events, regions, semaphores, and mailboxes lets you synchronize multiple signals and threads and simulate complex systems with greater ease.
- Because Vera is designed to verify existing hardware description models, it can interact with both Verilog and VHDL designs. This means that engineers can create testbenches in Vera that will interact with existing Verilog and VHDL code.
- The modular approach used by Vera makes the use of multiple test benches easy and efficient. Changes to the hardware design do not require recompilation of Vera testbenches. Further, changes to a testbench require that only the changed testbench be recompiled.

1.4 Overview

This section explains the basics of Vera and how it is used in a real-world context.

1.4.1 How Vera Is Used

Vera is typically used to develop regressible, self-checking tests for hardware designs. Vera test benches are created and linked to existing HDL hardware models using Vera's interface construct. Then the Vera driver provides all or most of the stimuli to the device under test (DUT). These stimuli simulate input that would be generated by other units in the real system. Thus, Vera's role is fundamentally to simulate the external systems that would drive the device in a real environment.

Figure 1-1 diagrams Vera's basic usage model.

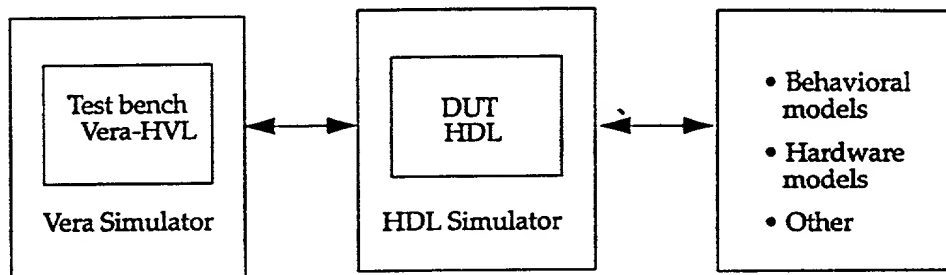


Figure 1-1 Typical Vera-CS usage model

Behavioral models can also be added where appropriate. Memory models and other highly-algorithmic modules are good candidates for behavioral implementations rather than mimicking the features with Vera code. Such models can be written in the HDL or Vera (depending on the required functionality).

1.4.2 Vera Components

The Vera verification suite includes Vera-HVL, the verification language and language compiler, the Vera virtual machine, which connects to Verilog and VHDL, and Vera-CS, the cycle-based simulator. The simulator can be used as a stand-alone simulator, but is most often used in conjunction with Verilog and VHDL designs.

The Vera virtual machine runs with the Verilog and VHDL simulators, using user-defined interfaces to convey signal values between the Vera and HDL domains. Each interface is defined by two main elements: the signals used to drive and sample HDL nodes and a clock that specifies all signal timing. The clock determines when signals are driven and sampled.

Simulations are governed by clock edges. All events occur at rising and falling clock edges (+/- any skew). As the HDL simulators detect these clock edges, they pass control to Vera. Then Vera runs its code non-preemptively until it reaches an HDL interface primitive such as drive or expect. To execute the primitives, Vera passes control back to the HDL. Because the Vera code is run non-preemptively, no simulation time passes while the Vera code executes.

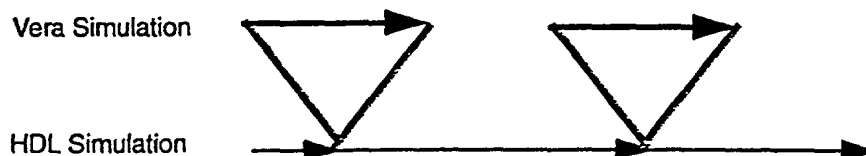


Figure 1-2 HDL and Vera Execution

In addition to being able to run with the HDLs, Vera-CS can run with functional circuit models written in C or other languages.

1.4.3 Vera/HDL Flow of Data

It is helpful to identify three major regions in the Vera/HDL environment: the Vera domain, the HDL domain, and the Vera/HDL interface. Each region has several key components.

The Vera Domain

In the Vera domain, there is Vera-HVL, the Vera virtual machine, and a Vera compiler. The Vera compiler compiles code written in Vera-HVL and translates it into an intermediate form used by the simulator. This intermediate form (.vro object files) can be generated from both the main Vera program and any subprograms that may be included. Note that they can be compiled separately and linked at simulation time.

The HDL Domain

In the HDL domain, there is the hardware description model coded in the HDL and the HDL simulator. The HDL simulator acts in conjunction with the Vera simulator across the Vera/HDL interface.

The Vera/HDL Interface

The Vera/HDL interface is the most important region of the Vera/HDL environment. It includes an interface construct, an HDL shell, global variables, and HDL tasks that will be called within Vera.

The interface construct maps signals from the HDL domain to the Vera domain. It specifies which signals will be observed and controlled, and determines the flow of data back and forth between the two domains.

The HDL shell is generated by the Vera compiler. It instantiates the interfaces via PLI calls.

The global variables and HDL tasks are the shared variables and tasks that are used in both domains.

Figure 1-3 shows a schematic of the flow of data within the Vera/HDL environment.

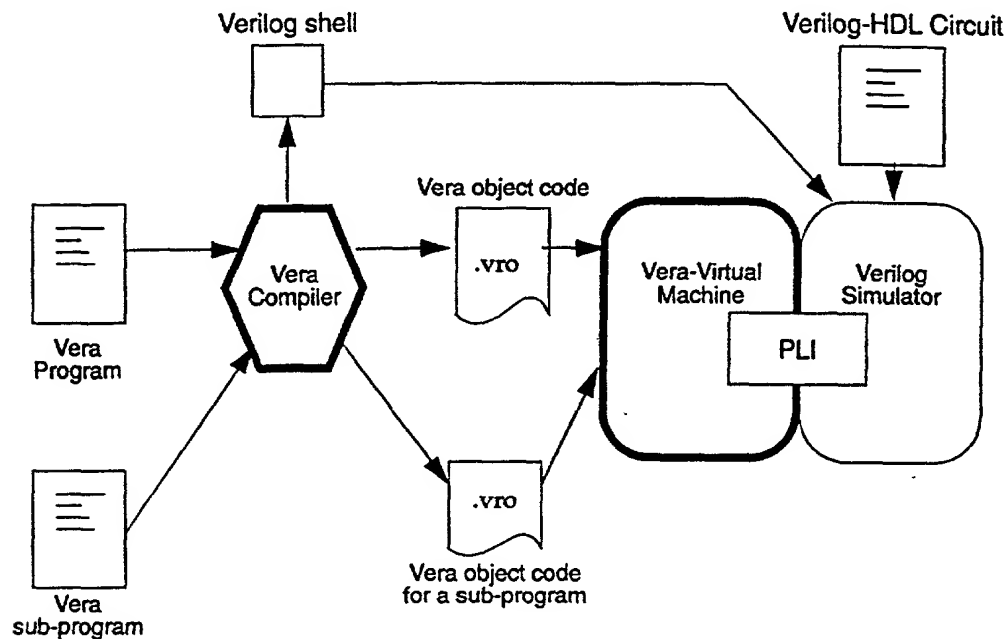


Figure 1-3 Vera Environment Flow

1.5 A Small Example

To understand how Vera can be used, we follow the process of verifying a simple Verilog module in this chapter. See the tutorial for additional examples (\$VERA_HOME/tutorial). The following Verilog module is a 2-input round-robin arbitrator.

```
module rrarb (request, grant, reset, clk);
  input  [1:0] request;
  output [1:0] grant;
  input  reset;
  input  clk;
  wire  winner;
  reg last_winner;
  reg  [1:0] grant;
  wire  [1:0] next_grant;

  assign next_grant[0]
    = ~reset & (request[0] &
      (~request[1] | last_winner));
```

```

assign next_grant[1]
  = ~reset & (request[1] &
    (~request[0] | ~last_winner));

assign winner
  = ~reset & ~next_grant[0] &
    (last_winner | next_grant[1]);

always @(posedge clk) begin
  last_winner = winner;
  grant = next_grant;
end

endmodule

```

The Verilog and Vera source code for this example are available in \$VERA_HOME/samples/arbiter. Running the Vera template generator (**vera -tem**) creates a simulation environment and a test template. The following command line will generate three files, `rrarb.test_top.v`, `rrarb.if.vrh` and `rrarb.vr.tmp`:

```
vera -tem -t rrarb -c clk rrarb.v
```

In the command line, `-t` specifies the name of the module to be tested, `-c` specifies the clock input (optional), and the last argument, `rrarb.v`, specifies the Verilog file name.

The first file `rrarb.test_top.v` generated by **vera -tem** is the Verilog simulation top file:

```

module rrarb_test_top;
  parameter simulation_cycle = 100;

  reg SystemClock;
  wire [1:0] request;
  wire [1:0] grant;
  wire reset;
  wire clk;
  assign clk = SystemClock;

  vera_shell vshell(
    .SystemClock(SystemClock),
    .rrarb_request(request),
    .rrarb_grant(grant),
    .rrarb_reset(reset),
    .rrarb_clk(clk)
  );

  rrarb dut (
    .request(request),
    .grant(grant),
    .reset(reset),

```

```

        .clk(clk)
    );

    initial begin
        SystemClock = 0;
        forever begin
            #(simulation_cycle/2)
            SystemClock = ~SystemClock;
        end
    end
endmodule

```

In the top file, the module `rrarb` is instantiated as `dut` (device under test) and connected to the `vera_shell` module. The Vera virtual machine uses this `vera_shell` module to interface to the Verilog world. A clock signal `SystemClock` is generated and connected to `vera_shell`'s `SystemClock`. This signal is used to count simulation cycles in the Vera world. The input `clk` is also driven by `SystemClock` since `-c` option was specified. Note that in systems with multiple clocks and multiple interfaces, `clk` and `SystemClock` can be independent.

The second file, `rrarb.if.vrh`, contains the interface and signal definition. This file is included in the third file, the template file, right before the program declaration.

```

interface rrarb {
    output [1:0]    request OUTPUT_EDGE OUTPUT_SKEW;
    input  [1:0]    grant   INPUT_EDGE;
    output          reset   OUTPUT_EDGE OUTPUT_SKEW;
    input          clk      CLOCK;
}

```

In this interface file, an interface `rrarb` is defined with signals on the `rrarb` module body. Output signals of the `rrarb` Verilog module are defined as inputs, and inputs of the Verilog module are defined as outputs in the Vera program interface, except for `clk`, which is driven by verilog top (since the `-c` option was specified).

The third file, `rrarb.vr.tmp`, is the Vera program template file:

```

#include <vera_defines.vrh>
#define OUTPUT_EDGE  PHOLD
#define OUTPUT_SKEW  #1
#define INPUT_EDGE    PSAMPLE
#include "rrarb.if.vrh"

// tasks/functions

program rrarb_test

{ // start of top block

    // global variables

    // start of test program

```

```

// Example of drive:
// rrarb.request = 0;

// Example of expect:
// rrarb.grant == 0;

} // end of program rrarb_test

```

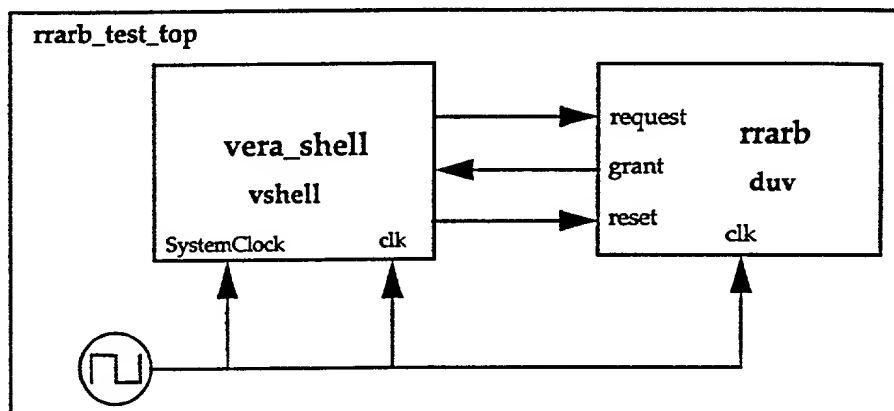


Figure 1-4 Top level connection

Using the template, we can easily write the test code. First, both requests are driven to 0 and `rrarb.reset` is asserted to initialize the internal states. Then grants are checked for each request. Finally, both requests are asserted simultaneously, and the round-robin control is checked.

```

#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>
#include "rrarb.if.vrh"

```

```

program rrarb_test
{
  // start of top block

  // reset
  rrarb.request = 2'b00;
  rrarb.reset = 1;
  @1 rrarb.reset = 0;

  // simple requesting
  @1 rrarb.request = 2'b01;
  @1 rrarb.grant == 2'b01;
  rrarb.request = 2'b00;
}

```

```

@1 rrarb.request = 2'b10;
@1 rrarb.grant == 2'b10;
rrarb.request = 2'b00;

// round-robin test
@1 rrarb.request = 2'b11; // requesting both port
@1 rrarb.grant == 2'b01; // should get grant 0, since last was 1
@1 rrarb.grant == 2'b10; // keep asserting both port, get gnt 1
rrarb.request = 2'b00; // negate

} // end of program rrarb_test

```

This command line compiles the file:

```
vera -cmp rrarb.vr
```

The Vera compiler will compile the `rrarb.vr` and generate two files: `rrarb.vshell`, a Verilog file that contains the `vera_shell` module, and `rrarb.vro`, the Vera object file.

Now run the simulation using Vera with Verilog-XL by typing:

```
verilog rrarb.test_top.v rrarb.v rrarb.vshell +vera_load=rrarb.vro
```

You see a Verilog simulation message such as:

```

VERILOG-XL X.X   MMM DD, YYYY HH:MM:SS
* Copyright Cadence Design Systems, Inc. YYYY, YYYY. *
Compiling source file "rrarb.test_top.v"
Compiling source file "rrarb.v"
Compiling source file "rrarb.vshell"
Highest level modules:
rrarb_test_top

Vera: finish encountered at time      850 cycle      9
      total      mismatch: 0
           vca_error: 0
      fail(expected): 0
           drive: 9
           expect: 4
           sample: 0
           sync: 0

Type ? for help
C1>

```

1.6 Multiple-Clock / Multiple-Module Systems

Vera supports complex clocking with any number of derived or independent clocks as well as asynchronous interactions that fall outside the scope of clocking. Understanding the concepts in this section is critical in handling system testing and multiple clocks.

Vera's `SystemClock` gives Vera a notion of time (cycle number). There are several operations that use this as a convenient reference signal. In most cases, it makes sense to tie the `SystemClock` to the actual system clock, but in others it might be offset or derived from another reference.

Most operations are tied to the timing of the actual signals. Independently of `SystemClock`, you can define any number of clocking domains (a "Vera interface"), each with its own clock reference.

In simple systems with just one interface, the real system clock, `SystemClock`, and the interface clock might be tied together, or just derived from one common signal by assigning different delays. Vera itself imposes no restrictions on the relation between the real system clock (Vera's reference) and the clocking for each interface.

Hence, interfaces in Vera are related to the timing of signals as opposed to the boundaries of modules or any such implementation-related issue. When you define an interface, you assign to it a clock reference.

Note that you cannot have multiple clocks on a single interface because the clock defines a clocking domain (interface). Instead, split your signals across multiple interfaces, each one with its own clock.

Sometimes the question arises "can I have multiple clocks in an interface?" The answer is: "no, it doesn't make sense - the clock defines a clocking domain (interface) - what you probably want is to split your signals across multiple interfaces, each one with its own clock."

As a final note, driving of synchronous signals is statically tied to a given clocking domain for the duration of a testbench. However, sampling can be done in multiple clocking domains by connecting a Verilog signal to multiple Vera interfaces.

1.7 Simple Example of a 2-Clock System

In this example we will have a system in which there is a clocking module that generates two main clocks (`clk1` and `clk2`). The system contains two major portions, each one running from its own clock.

```
// ...
#define TIMING_A PSAMPLE #-1 PHOLD #6
#define TIMING_B PSAMPLE #-2 PHOLD #5
#define TIMING_C NSAMPLE #0// sample on falling edge
#define TIMING_D PHOLD #3

// connect signals to clocking domain 1
interface subsys1 {
    inout [31:0] data TIMING_A verilog_node "sys.a.subsys1.D[31:0]";
```



```

output [63:0] addr TIMING_D verilog_node "sys.b.subsys1.A[63:0]";
input clk CLOCK verilog_node "sys.clk_gen.clk1";
input str TIMING_C verilog_node "sys.str";
// ....
}

// connect signals to clocking domain 2
interface subsys2 {
  inout [16:0] data TIMING_B verilog_node "sys.subsys2.D[16:0]";
  output [32:0] addr TIMING_D verilog_node "sys.subsys2.A[32:0]";
  input clk CLOCK verilog_node "sys.clk_gen.clk2";
  input str TIMING_C verilog_node "sys.str";
  // ....
}

// connect Vera's SystemClk to "sys.sysclk"
verilog_node CLOCK "sys.sysclk";
//...

```

In this example, Vera will drive `subsys1.data` 6 time units after the rising edge `subsys1.clk` (same as `sys.clk_gen.clk1`) and sample it 1 time unit before the rising edge.

The `SystemClock` (the same as `sys.sysclk`) is used only in relation to calls such as `get_cycle()`, or operations as `@(CLOCK)`. All other sampling, driving, synchronization, etc., operations involve specific signals, so they use the timing specified in the corresponding interface clocks (rather than `SystemClock`).

Note that the Verilog node `sys.str` is being observed in both timing domains (as `subsys1.str` and `subsys2.str`, respectively). The only difference is that in each domain we are using a different clock reference and a different offset within the clock. You can use multiple clock references to sample a signal, but you must choose one clock reference to drive.

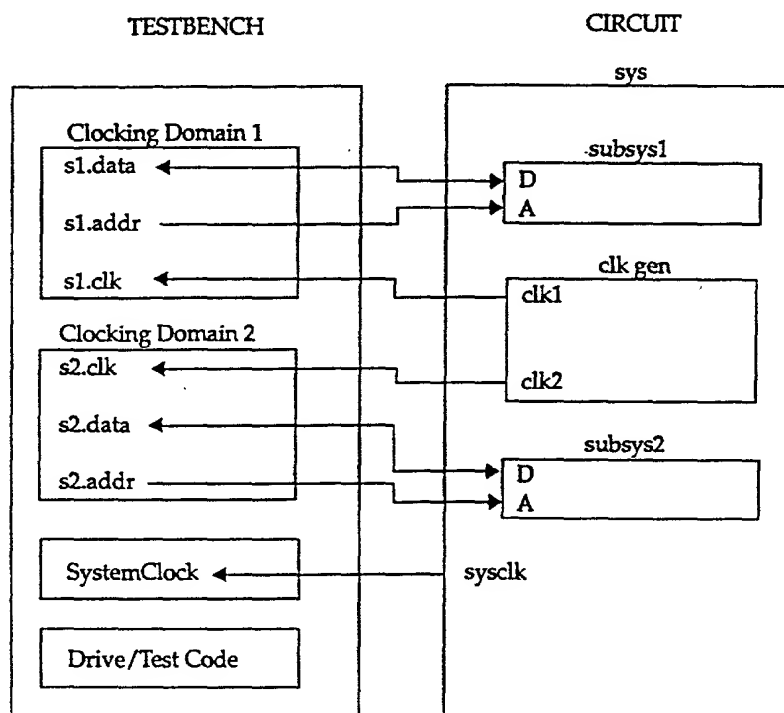


Figure 1-5 Multiple Modules and Multiple Clocks

There are no restrictions in terms of how each one of these clocks are related. In fact, they can even stop and continue independently. Vera only tracks their edges. At that point (plus or minus any skew), the signals related to the clock that changed may be driven, sampled, etc. (depending on what the test code may be doing).

2. Installation and Setup

This chapter details the installation and setup of the Vera verification suite. It includes these sections:

- Installation
- Licensing
- Running Vera with Verilog-XL
- Running Vera with Chronologic VCS
- Running Vera with the Verilog ModelTech Simulator
- Running the Vera Stand-Alone Simulator
- Testing the Installation
- Vera Support

2.1 Installation

1. Create a release directory on the machine you want to act as your Vera server.
2. Download the Vera release to the release directory you created. The ftp address is provided by Synopsys. Email vera-support@synopsys.com for help.
3. Set the environment variable `VERA_HOME` pointing to the Vera release directory:

```
setenv VERA_HOME <directory_path>
```
4. Add `$VERA_HOME/bin` to your search path.

2.2 Licensing

Vera 3.1 and higher uses the industry-standard FlexLM licensing system.

2.2.1 Licensing for Current FlexLM Users

If you use FlexLM, check that your Vera key is for the same server(s) that show in your current `$LM_LICENSE_FILE`, add the Vera feature to your license file, and then add the daemon line for `$VERA_HOME/bin/ssilmd`.

2.2.2 Licensing for Other Users

Vera uses the industry-standard FlexLM licensing system, which provides floating licenses for a networked environment. To license your version of Vera, you need to do 4 things:

1. Obtain your Server-ID.
2. Edit your Flex license file to add the absolute path to the Vera verification tool (\$VERA_HOME) and the Synopsys license daemon (\$VERA_HOME/bin/ssilmd).
3. Set LM_LICENSE_FILE to point to the license file.
4. Start the FlexLM license server.

For more information, use any Web browser to view the HTML data in:

`$VERA_HOME/doc/flexlm/man.html`

Getting the Server-ID (lmhostid)

Flex network floating licenses are managed from one server (or more, when redundant servers are set up). When you purchase Vera licenses you need to specify the server(s) for which the keys should be hosted, including both their name and their Flex-ID. To get the Flex-ID, type the following from a Unix shell:

For SPARC servers: `hostid`

For all other servers: `lmhostid`

The `lmhostid` executable can be found within the Vera release, in:

`$VERA_HOME/bin/lmhostid`

or on the Synopsys FTP site:

`ftp.systems.com`

To use the FTP site:

1. Use your FTP tool to connect to the Synopsys FTP site. Log in with the username *anonymous* and password *anonymous*.
2. Change to the directory designated for your server type:

`cd /pub/ssi/<server_type>`

where `<server_type>` is `sunos`, `solaris`, `hpux`, `aix`, `SGI`, or `ALPHA`.

3. Make sure you are in binary mode. Download the zipped executable:

`get lmhostid.gz`

4. Download the gzip tool if your site does not already have it:

`get gzip`

5. After exiting FTP, make sure the zipped executable is in the directory you want (`$VERA_HOME/bin/lmhostid` is preferable). Then decompress the zipped file:

`gzip -d lmhostid.gz`

6. Set the permissions on the executable:

```
chmod a+x lmhostid
```

FLEXlm License File Contents

The FLEXlm license file contains 3 important pieces of information:

1. Server line:

```
SERVER <server_name> <server_lmhostid> <port>
```

Make sure the key you got is for the proper hostid/hostname. There should be only one server line per Flex license file.

For example:

```
SERVER mars 12345678 1700
```

2. Daemon line(s):

```
DAEMON ssilmd <absolute_path_to_ssilmd_binary>
```

You must use the absolute network path **WITHOUT** environment variables to a copy of the Synopsys license daemon found in `$VERA_HOME/bin/ssilmd`. In a given FlexLM file there can be 1 daemon line for each vendor.

For example:

```
DAEMON ssilmd /net/jupiter/home/cad/ssi/vera/bin/ssilmd
```

3. Feature line(s):

```
FEATURE <tool> <daemon> <version> <expires> <num> <key>
```

There will be one feature in your LM license file for each tool.

For example:

```
FEATURE vera ssilmd 3.000 31-dec-96 2 BB6E7071B0C8C9877168
```

LM_LICENSE_FILE Environment Variable

You tell FlexLM where your keys are by setting an environment variable as follows:

```
setenv LM_LICENSE_FILE <path_to_your_license_file>
```

Each license file may contain licenses for many packages and from multiple vendors. It is simplest to have all your licenses in one file. However, you can specify multiple files as follows:

```
setenv LM_LICENSE_FILE <path_to_file1>:<path_to_file2>:<path_to_fileN>
```

Add this command to your `.login` or `.cshrc` files to ensure that it is executed upon logging in.

Starting the Flex License Manager

If it is not already running on the host server, start the license manager by typing from the Unix command line:

```
$VERA_HOME/bin/lmgrd -c <vera_license_file> >& /tmp/flexlmgrd.log
```

If it is already running and you want Flex to know about your update to the license file, enter:

```
$VERA_HOME/bin/lmreread
```

Note – Use "lmgrd" version 5.0 or higher. If you are using an older version supplied by another tools vendor, you may need to upgrade to the "lmgrd" supplied with Vera (which would also provide the backwards compatibility for tools that use older FlexLM versions). To find out the version of a given lmgrd, enter `lmgrd -v` at the command line.

Versions Enabled and Backwards Compatibility

License keys for higher versions of the software enable lower versions as well. Also, Vera licenses ignore any decimals in the version number except for the 1st one (e.g., a key for version 4.40 will enable, vera-3.1, vera-4.0, and vera-4.49, but not vera 4.5).

Versions 3.0 or lower did not use FlexLM. You can run in your environment a mix of Flex and non-Flex based Vera versions. However, beware that the license files, servers, and environment variables are unrelated and cannot be combined. You can have both the old SSI license manager and Flex running, have both the old SSI_LICENSE and LM_LICENSE_FILE environment variables set. As long as your search path is set to pick the right Vera/HDL versions, you should have no problems.

2.2.3 Controlling Vera License Queuing

The environment variable `VERA_WAIT_LICENSE` allows you to control the checking out and queuing order of full and runtime Vera licenses. The syntax from the Unix command line is:

```
setenv VERA_WAIT_LICENSE value
```

value - The *value* can be either 1, which queues up for runtime if both full and runtime licenses are unavailable, or 2, which polls for either license every 10 seconds if both full and runtime licenses are unavailable. All other values (or if `VERA_WAIT_LICENSE` is not set) terminate the simulation if neither license is available.

Note – The environment variable `SSI_WAIT_LICENSE` must not be set.

2.3 Running Vera with Chronologic VCS

You must generate a new Chronologic VCS `simv` executable with each new hardware description. To link it to Vera, you need to invoke `vcs` with the following arguments:

a) your verilog model(s)

b) option "-P \$VERA_HOME/lib/vera_pli.tab" to specify the PLI table (or merge this PLI table with other ones you may already be using)

c) the *\$VERA_HOME/lib/libSysSciTask.a* library

For example, to generate a new *simv* executable for a design *model.v*, enter:

```
vcs model.v -P $VERA_HOME/lib/vera_pli.tab $VERA_HOME/lib/libSysSciTask.a
```

If you are using SunOS, you need to run *ranlib* before linking the *libSysSciTask.a* library:

```
ranlib $VERA_HOME/lib/libSysSciTask.a
```

If you are using HPUX, you must invoke the following options in addition to those specified previously:

```
-M -gen_c -LDFLAGS "-E -a archive_shared" -ldld
```

For example, to generate a new *simv* executable for a design *model.v*, enter:

```
vcs model.v -Mupdate=1 -gen_c -P $VERA_HOME/lib/vera_pli.tab \
$VERA_HOME/lib/libSysSciTask.a -LDFLAGS "-E -a archive_shared" -ldld
```

2.4 Running Vera with Verilog-XL

To use Vera with Verilog-XL, you must link the *libSysSciTask.a* library into your Verilog executable. You must do this only when first using Verilog-XL with Vera.

Note – Version 3.1p4 of Vera and later versions dynamically load the *vera.dl* library. If you've previously linked your Verilog-XL with *libSysSciTask.a*, you **DO NOT** need to relink it. Just change your *VERA_HOME* environment variable, and Verilog-XL will automatically use the *vera.dl* library in the new Vera release.

Environment

You must have the Cadence binaries and the C-compiler in your search path. To make sure they are set correctly, add the following to your *.cshrc* file:

```
# tools
setenv CDS_INST_DIR <path_to_CADENCE_install_directory>/tools.<arch>
setenv C_COMPILER_HOME<path_to_C_compiler_install_directory>
#
# search path
set path = ($path $CDS_INST_DIR/bin $C_COMPILER_HOME/bin)
```

veriusers.c

The file `veriusers.c` contains information that Verilog uses to link in user-defined system tasks. You must include these files in your `veriusers.c` file:

```
vmc_veri_funext.h
vmc_veri_fundef.c
```

If you have other tasks that have to be linked in, such as PowerFault-IDDQ's, you need to combine what is included in `$VERA_HOME/lib/veriusers.c` with the `veriusers.c` for the other tasks.

vconfig

The `vconfig` procedure creates a `cr_vlog` script, which is used to link Vera into the new Verilog executable. To run `vconfig`:

1. Begin `vconfig`:

```
vconfig
```

If you get a message saying: "vconfig: Command not found", it means that your search path is missing Cadence's `$CDS_INST_DIR/tools/bin` directory.

2. At the following prompt:

The user template file 'veriusers.c' must always be included in the link statement. What is the path name of this file?

Enter the path to your own `veriusers.c`.

3. At the following prompt:

List the files one at a time, terminating the list with a single '.'

Enter:

```
./libSysSciTask.a
```

Then specify any other libraries you may need to link in (1 per line). After all the libraries have been specified enter a single period (.) on a new line.

The `vconfig` program will create a script named `cr_vlog` (or whichever name you choose when prompted).

Before Running cr_vlog

If you are using HPUX 9.0 or HPUX 10.2, your `cr_vlog` must use the "-Wl,-E" compile option. Change the "cc" command from:

```
cc -o verilog
```

to:

```
cc -Wl,-E -o verilog
```


If you are using either Verilog-XL version 2.X OR you are using HP-UX 9.0, you must also link with the "-ldld" library. Check that one of the last lines of your *cr_vlog* file matches the following:

```
+O3 -lm -lBSD -lcl -N -ldld
```

If you are using SunOS, you should run *ranlib* before running *cr_vlog*:

```
ranlib sunos4.1/libSysSciTask.a
```

If this step is omitted, you will get link complaints when you invoke *cr_vlog*.

If you are using Solaris, your *cr_vlog* script must link with these libraries:

```
-lsocket -lnsl -lintl
```

Check that these libraries are included in your *cr_vlog* script.

Running *cr_vlog*

When you run the *cr_vlog* script you will compile a new Verilog executable with the *libSysSciTask.a* library. To run it, enter:

```
cr_vlog
```

Install the new Verilog executable you created the directory you want to run it from. Note that if you chose to name the new executable "verilog", you should have \$VERA/bin ahead of \$CADENCE/tools/bin in your search path, so that the right version of Verilog is picked up.

2.5 Running Vera with the Verilog ModelTech Simulator

You must recompile your MTI simulator and link it with Vera before running any simulations. To recompile your ModelTech simulator:

1. Compile *veriusers.c* for the MTI simulator.

For Solaris, type:

```
cc -K pic -DMTI -DaccVersionLatest -c -I<modeltech_include_dir> \
-I$VERA_HOME/lib veriusers.c
```

OR

```
gcc -fpic -DMTI -DaccVersionLatest -c -I<modeltech_include_dir> \
-I$VERA_HOME/lib veriusers.c
```

Then type:

```
ld -G -o veriusers.so veriusers.o libSysSciTaskpic.a
```

For SunOS, you must run *ranlib* *libSysSciTask.a* prior to compiling. Then type:

```
cc -DMTI -DaccVersionLatest -c -I<modeltech_include_dir> \
-I$VERA_HOME/lib veriusers.c
```

```
ld -o veriuser.so veriuser.o libSysSciTaskpic.a
```

For HP/UX, type:

```
cc -DMTI -DaccVersionLatest -c +z -I<modeltech_include_dir> \
-I$VERA_HOME/lib veriuser.c
```

OR

```
gcc -fpic -DMTI -DaccVersionLatest -c -I<modeltech_include_dir> \
-I$VERA_HOME/lib veriuser.c
```

Then type:

```
ld -b -o veriuser.sl veriuser.o libSysSciTaskpic.a -lc
```

2. Set up ModelTech to load veriuser.so (veriuser.sl for HP).

Edit vsystem.ini or modelsim.ini to include:

```
[vsim]
Veriuser = $VERA_HOME/lib/verisure.so
```

OR

```
setenv PLIOBJS $VERA_HOME/lib/veriuser.so
```

OR

```
-pli $VERA_HOME/lib/veriuser.so
```

2.6 Running the Vera Stand-Alone Simulator

Vera can be used by itself as a concurrent programming language. As such, it is a cycle-based simulator, which can run Vera models stand-alone or linked with additional C models. The pre-compiled copy of this simulator is in *\$VERA/home/bin/vera_cs*. If you want to link in your own C code to this simulator, instead of using the pre-compiled copy above, you must compile *vera_user.c* and link it with *\$VERA_HOME/lib/libVERA.a*. Additionally, if you are using a Solaris platform you must link in *-lsocket -lnsl -lintl* along with any other appropriate libraries. See Section 18.8, "Running Vera Stand-alone" for usage of the resulting simulator.

Note – If you do not use Verilog-XL and do not have the *veriuser.h* file available, you must define "NO_VERILOG_SIM" from the command line when you compile *vera_user.c*.

2.7 Testing the Installation

Make sure that the Verilog executable in your path is called `verilog` and that it is linked with the Vera tasks. In `$VERA_HOME/samples` there are complete examples, each with a `doit` script. To test the installation, check that the scripts compile the tests and run Vera with Verilog.

2.8 Vera Support

Please use the following email addresses for Vera-related issues:

Technical Issues:

Customer Support (setup, bugs, help): vera-support@synopsys.com

Enhancement Requests: vera-enhance@synopsys.com

Documentation Issues: vera-doc@synopsys.com

Other Feedback: vera-feedback@synopsys.com

Information, Sales, and Licensing:

Licensing: vera-license@synopsys.com

Sales Information: vera-sales@synopsys.com

Copyright © 2000 Synopsys, Inc. All rights reserved. Synopsys, the Synopsys logo, and Vera are registered trademarks of Synopsys, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

3. Vera-HVL: The Language

This chapter describes the basics of Vera-HVL. It introduces the lexical conventions and some of the basic components of the language. It has these sections:

- Lexical Conventions
- Data Types and Variable Declaration
- Arrays
- Strings
- Enumerated Types
- Vera Operators
- Variable Assignment

3.1 Lexical Conventions

Vera-HVL source code consists of a stream of lexical elements. The types of lexical elements are:

- White space
- Comments
- Statement Blocks
- Identifiers and Keywords
- Strings
- Numbers

3.1.1 White Space

White space is any sequence of spaces, tabs, newlines, and formfeeds. White space is used in Vera as token separators. Whitespace can be used freely and is ignored by Vera, except within a string.

3.1.2 Comments

Vera supports two forms of comments: a one-line comment and a block comment.

A one-line comment starts with a double slash (//) and finishes out the line. The syntax is:

```
any_vera_statement; // One line comment.
```

A block statement starts with a /* and ends with a */. Everything between the start and end tags is a comment. The syntax is:

```
/* Blocks of comments
   can take up
   multiple lines */
```

Note – Block comments cannot be nested.

3.1.3 Statement Blocks

Vera supports three methods of creating statement blocks: begin/end, braces, and fork/join.

The syntax for begin/end statement blocks is:

```
begin
    vera_statements
end
```

The syntax for statement blocks using braces is:

```
{
    vera_statements
}
```

The syntax for fork/join statement blocks is:

```
fork
    process1();
    process2();
    ...
    processN();
join
```

Forks and joins are discussed in more detail in Section 6.1, "Fork and Join."

3.1.4 Identifiers and Keywords

An identifier is a sequence of letters [a-zA-Z], digits[0-9] and underscores[_]. Identifiers are case-sensitive and cannot begin with a digit.

In addition to the Vera keywords, all signal types should be considered keywords. Table 3-1 lists the Vera signal types.

Table 3-1 Vera Signal Types

ASYNC	CLOCK	NDRIVE	NHOLD
NRX	NRZ	NR0	NR1
NSAMPLE	PDRIVE	PHOLD	PRX
PRZ	PR0	PR1	PSAMPLE

Vera also uses a set of predefined q_values with its Value Change Alert implementation (Section 4.4.5, "Value Change Alert (VCA)"). Table 3-2 lists the predefined q_values.

Table 3-2 VCA q_values

gnr	grx	grz	grl
gr0	nr	rx	rz
r0	r1	snr	srx
srz	sr0	sr1	

Vera has several predefined system functions and system tasks whose names must not be used as program identifiers. Table 3-3 lists the predefined identifiers.

Table 3-3 Vera Predefined System Functions and Tasks

alloc	call_func	call_task	cast_assign	close_conn
delay	error	error_mode	exit	fclose
fflush	flag	fopen	fprintf	freadb
freadh	freadstr	get_bind	get_bind_id	get_conn_err
get_cycle	get_plus_arg	get_systime	get_time	mailbox_receive
mailbox_send	make_client	make_server	printf	rand48
random	region_enter	region_exit	rewind	semaphore_get
semaphore_put	sprintf	sscanf	stop	sync
timeout	trace	trigger	unit_delay	up_connections
urand48	urandom	vsv_call_func	vsv_call_task	vsv_close_conn
vsv_get_con_err	vsv_make_client	vsv_make_server	vsv_up_connections	vsv_wait_for_done
vsv_wait_for_input	wait_child	wait_var		

Vera has several predefined system macro constants whose names must not be used or redefined without proper system tasks. Table 3-4 lists the predefined constant identifiers.

Table 3-4 Vera Predefined Constants

stderr	stdin	stdout	ALL
ANY	BAD_STATE	BAD_TRANS	CHECK
CHGEDGE	CLEAR	CROSS	CROSS_TRANS
DEBUG	DELETE	EC_ARRAYX	EC_CODE_END
EC_CONFLICT	EC_EXPECT	EC_FULLEXPECT	EC_MBXTMOUT
EC_NEXPECT	EC_RETURN	EC_RHNTMOUT	EC_SCONFLICT
EC_SEMTMOUT	EC_SEXPECT	EC_SFULLEXPECT	EC_SNEXPECT
EC_USERSET	EQ	EVENT	FIRST
GE	GOAL	GT	HAND_SHAKE
HI	HIGH	HNUM	LE
LIC_EXIT	LIC_PRERR	LIC_PRWARN	LIC_WAIT
LO	LOAD	LOW	LT
MAILBOX	NAME	NEGEDGE	NEXT
NO_OVERLAP	NO_WAIT	NUM	NUM_BIN
OFF	ON	ONE_BLAST	ONE_SHOT
ORDER	POSEDGE	PROGRAM	RAWIN
REGION	REPORT	SAVE	SEMAPHORE
SET	SILENT	STATE	STR
STR_ERR_OUT_OF_RANGE	STR_ERR_REGEXP_SYNTAX	SUM	TRANS
VERBOSE	WAIT		

3.1.5 Strings

A string is a sequence of characters enclosed by double quotes. A string must be contained in a single line unless the new line is immediately preceded by a back slash. In this case, the back slash and new line are ignored.

3.1.6 Numbers

In Vera, a number can be formed using either the **LIT_INTEGER** or **NUMBER** format.

The **LIT_INTEGER** format is a simple decimal number specified as a sequence of digits from 0 to 9. Negative signs are allowed to specify negative integers. Underscores are ignored and may be used for clarity. The syntax is:


```
[0123456789]+
```

The **NUMBER** format takes these forms:

```
<size>'<base><number>
```

<size> - The **<size>** specifies the number of bits in the number. The size upper limit is dependent on the host machine, but it is generally 32 bits. If the **<size>** is omitted, the number of bits for **<number>** defaults to the host machine word size. A plus or minus sign before the **<size>** specification signifies the number's polarity.

<base> - The **<base>** is always preceded by a single quote ('). The **<base>** can be one of the following: d(ecimal), h(exadecimal), o(ctal), or b(inary). The base identifier can be either upper or lower case.

<number> - The valid elements of **<number>** for each **<base>** are:

```
'b (binary): [01xXzZ_]
'd (decimal): [0123456789_]
'o (octal): [01234567xXzZ_]
'h (hexadecimal): [0123456789abcdefABCDEFxXzZ_]

```

X and x represent unknown values, and Z and z represent high impedance values in binary, octal, or hexadecimal form. Underscores are ignored.

If the most significant specified digit of a **<number>** representation is an x or a z, the Vera compiler extends the x or z to fill the higher order bits or digits. For example, 8'bx means 8'bxxxxxxx, or 8'bz00 means 8'bzzzzzz00.

3.2 Data Types and Variable Declaration

Vera's basic data types are integers, bits, strings, events, bind_var, classes, and enumerated types. You can also make composite types of the base types with arrays, associative arrays, and records.

Variables are seen globally if they are declared at the top (program) level. If they are declared in blocks (begin/end, {, }. fork/join), they are seen locally.

3.2.1 Integers

Integers are signed variables. They can have integral values between $1-2^{31}$ and $2^{31}-1$. An integer may become X (unknown) when it is not initialized or when an undefined value is stored.

The syntax to declare an integer is:

```
integer variable_name = initial_value;
```

The initialization is optional in the variable declaration.

Note – For expressions involving both bit and integer types, the integer types are first converted to 32-bit unsigned integers.

3.2.2 Bits

Bits can have the value 0 (logic 0), 1 (logic 1), z (high impedance), or x (unknown).

The syntax to declare a bit is:

```
bit variable_name = initial_value;
```

The initialization is optional in the variable declaration.

Vera also supports bit fields. The syntax to declare a bit field is:

```
bit [high:low] variable_name = initial_value;
```

High specifies the upper limit on the field, and *low* specifies the lower limit on the field. The maximum size of a bit field is 4095 bits. When declaring bit fields, you cannot use variables for *high* and *low* specifiers.

Vera also supports registers. Registers in Vera are identical to bit vectors. The syntax to declare a register is:

```
reg [high:low] variable_name = initial_value;
```

3.2.3 Strings

Strings are character data types that have a wide range of operators associated with them for manipulating characters.

The syntax to declare a string is:

```
string variable_name = initial_value;
```

The initialization is optional in the variable declaration.

Strings and string operators are discussed in more detail in Section 3.4, "Strings."

3.2.4 Bind_vars

Bind_var variables hold binds. Binds are used to group and associate signals with ports. Bind_vars are used with task calls to specify which port the task call communicates with. For more information concerning binds, see Section 4.3.3, "Bind."

The syntax to declare a bind_var is:

```
bind_var variable_name = initial_value;
```

The initialization is optional in the variable declaration.

Bind_vars are discussed in more detail in Section 4.3.4.1, "bind_var Variables."

3.2.5 Events

Events are variable types used to manage triggers. They exist in one of three states: OFF, ON, and null. Events are passed as arguments to task, function, and method calls to specify the trigger point. They are used mainly to synchronize concurrent processes. Synchronization and the use of events is discussed in more detail in Section 6.2, "Events."

The syntax to declare an event is:

```
event variable_name = initial_value;
```

The default *initial_value* is OFF. You can only initialize events to null, which has the effect of the trigger always being ON.

3.2.6 Enumerated Types

Enumerated types are named integer constants. The syntax to declare an enumerated type is:

```
enum category = list;
```

OR

```
enum category {list}
```

The *category* is the name of the enumerated type. It is used to assign *list* values to variables.

List is a list of *category* values separated by commas. They are assigned sequential integer values in the order listed.

Enumerated types are discussed in more detail in Section 3.5, "Enumerated Types."

3.2.7 cast_assign()

Vera provides the `cast_assign()` system function to assign values to variables that might not ordinarily be valid because of typing. The syntax for `cast_assign()` is:

```
cast_assign(dest_var, source_exp [, check]);
```

dest_var - The *dest_var* is the variable to which the assignment is made. It can be any non-array scalar type (bits, integers, strings, enumerated types, bind_vars, port variables, events, and object handles).

source_exp - The *source_exp* is the source expression that is assigned to the destination variable.

check - The *check* modifier can optionally be specified using CHECK. Its use determines how the function handles invalid assignments.

When the `cast_assign()` system function is called without the CHECK modifier, the function assigns the source expression to the destination variable. If the assignment is illegal, a fatal runtime error occurs. When the `cast_assign()` system function is called with the CHECK modifier, the function makes the assignment and returns a 1 if the casting is successful, or it does not make the assignment and returns a 0 if the casting is unsuccessful. In the latter case, no runtime error occurs, and the destination variable is set to NULL, X, or uninitialized, depending on the data type.

Note - The compiler only checks that the destination variable and source expression are scalars. Otherwise, no type checking is done at compile time.

These are examples of the `cast_assign()` system function:

```
cast_assign(my_enum, 12*7);
cast_assign(my_portvar, my_bindvar);
```

The first example assigns the expression to the enumerated type. Without `cast_assign()`, this assignment is not allowed because of strong typing of enumerated types. The second example assigns the value of the `bind_var` to the port variable. Again, this assignment is not possible without `cast_assign()` because of strong typing of port variables.

A discussion of the `cast_assign()` system function and its use with objects is in Section 8.12, "Casting."

3.3 Arrays

Vera supports one-dimensional arrays, which are lists of variables that are all of the same type and called with the same name. Arrays in Vera can be static (global) or dynamic (local). You can also create associative arrays that have the advantage that each entry of the array gets allocated only when it is accessed.

3.3.1 Arrays

Any variable type can be declared as an array. The syntax to declare arrays is:

```
integer array_name[size];
bit [high:low] array_name[size];
bind_var array_name[size];
event array_name[size];
```

size - The *size* specifies the number of elements in the array. The maximum number of elements in an array is $2^{31}-1$ elements. For larger arrays, you should use associative arrays.

Accessing an array with an unknown bit ('x') in the index causes a simulation error. Also, writes to an array with an unknown in the index are ignored, and reads with an unknown in the index return 'X's.

Note that a bit field of an array element cannot be referenced directly. To reference a bit field of an array element, use a temporary variable. For example:

```
tmp = memory[42];
if (tmp[3:2] == 0) ...
```

You cannot initialize an array in the declaration.

3.3.2 Associative Arrays

Associative arrays are arrays whose dimensions are not specified. The syntax to declare associative arrays is:

```
integer array_name[];
bit [high:low] array_name[];
bind_var array_name[];
event array_name[];
```

Array elements in associative arrays are allocated dynamically, when you access a particular address. The array index tracks those elements that have been assigned values and stores those values within the array. When using integer and bit associative arrays, if you try to access an element that has not been assigned a value, an 'X' is returned. When using *bind_var* and event associative arrays, if you try to access an element that has not been assigned a value, a NULL value is returned.

Note - Using associative arrays slows down simulation time slightly, though the effect is usually unnoticeable.

Vera supports several mechanisms for analyzing associative arrays:

```
assoc_index (CHECK, array_name, index);
assoc_index (DELETE, array_name, index);
```

The *array_name* is the name of associate array you are analyzing.

The *index* is the numerical index of the element you are analyzing.

The **CHECK** function checks if an element exists at the specified index within the array. If it does, a 1 is returned. If it does not, a 0 is returned.

The DELETE function deletes the element specified at the specified index. If it is successful, a 1 is returned. If the element does not exist a 0 is returned.

Vera supports two additional functions used with associative arrays:

```
assoc_index (FIRST, array_name, index_var);
assoc_index (NEXT, array_name, index_var);
```

When the FIRST keyword is used, the function returns the element associated with the first valid index. The *index_var* is assigned the value of the first valid element in the array. The function returns a 1 if it is successful and a 0 if it fails.

When the NEXT keyword is used, the function returns the element associated with the next valid index. The *index_var* is assigned the value of the next valid element in the array. The function returns a 1 if it is successful and a 0 if it fails.

3.4 Strings

Within Vera, string data types are defined as classes. Vera also defines a set of internal variables and class methods used to analyze and manipulate strings.

3.4.1 Printing Strings

Vera strings are printed using the C-style printf function. The syntax to print a string is:

```
printf("string_format", variables);
```

Vera uses several predefined format specifiers. Table 3-5 lists Vera's format specifiers.

Table 3-5 Vera's Format Specifiers

Format Specifier	Format
%d or %D	a decimal number
%h or %H or %x or %X	a hexadecimal number
%o or %O	a octal number
%b or %B	a binary number
%c or %C	a single character
%s or %S	a string

The format specifiers are: %d or %D (decimal), %h or %H or %x or %X (hexadecimal), %o or %O (octal), %b or %B (binary). You can also use %s or %S to read in a string, or %c or %C to read in a single character.

By placing a number before the format specifier, you can specify how many digits are read in a particular number. If you want to minimize field width, use a 0 before the format specifier.

You can also use escape characters for formatting. Table 3-6 lists the Vera escape characters.

Table 3-6 Vera Escape Characters

Escape String	Character Produced
\n	new line
\t	tab
\\	\
\"	"
\ddd	a character specified in 1 to 3 octal digits
%%	%

3.4.2 Valid Operators

Vera provides a set of operators that can be used to manipulate combinations of string variables and string constants. Table 3-7 lists the valid operators.

Table 3-7 Valid Operators

Operator	Meaning
==	Check equality of two strings
!=	Check inequality of two strings
{str1,str2..}	Generate a concatenated string with <i>str1</i> , <i>str2</i> , ...
{num{str}}	Generate a string duplicated <i>num</i> times.

Note – You can compare string variables to null.

3.4.3 String Class Methods

Vera's string functions behave as class methods. Therefore, the string being analyzed must be specified in the method call. The syntax to make method calls is:

string_name.function()

string_name - The *string_name* is the string that is the target of the method call.

function - The *function* is the name of the method being called.

3.4.3.1 General String Class Methods

Vera provides a set of general functions (or methods) to handle hidden values within string variables.

len()

The `len()` function returns the length of the string as an integer. The syntax is:

```
string_name.len();
```

For example:

```
string str;
str = "This is a string";
printf("String length = %0d\n", str.len());
```

This example prints the string length of string *str*.

getc()

The `getc()` function returns a character at a specified location. The syntax is:

```
string_name.getc(i);
```

i - The *i* parameter must be an integer value.

The function returns the character specified by *i*. If the parameter is larger than the given string, 0 is returned.

For example:

```
string str = "This is a string";
printf("The fifth character is %c\n", str.getc(5)) ;
```

This example prints the fifth character in string *str*.is a bit function that

putc()

The `putc()` task assigns a given character to a specified location. The syntax is:

```
string_name.putc(i, "char");
```

i - The *i* parameter must be an integer value.

char - The *char* parameter must be a bit.

The task stores *char* in the *i* position within the string. If *i* is larger than the string, *char* is ignored and the internal flag `STR_ERR_OUT_OF_RANGE` is set.

For example:

```
string str = "X23456789";
str.putc(0, "1");
```


This example sets string *str* to "123456789".

get_status()

The `get_status()` function returns the current status flag value as an integer. The syntax is:

```
string_name.get_status();
```

The possible values are 1, which corresponds to `STR_ERR_OUT_OF_RANGE` (set by `putc`), and 2, which corresponds to `STR_ERR_REGEX_SYNTAX` (set by `match`). If there is no flag, the function returns a 0.

get_status_msg()

The `get_status_msg()` function returns a string describing the current status flag value. The syntax is:

```
string_name.get_status_msg();
```

The function returns the string `STR_ERR_OUT_OF_RANGE` (set by `putc`) or `STR_ERR_REGEX_SYNTAX` (set by `match`).

substr()

The `substr()` function returns the sub-string of characters between two specified locations. The syntax is:

```
string_name.substr(i, j);
```

i - The *i* parameter is the first location in the string.

j - The *j* parameter is the second location in the string.

The function prints the characters between the locations, inclusively. If the second argument is omitted, the function returns all characters from the first location to the end of the string.

For example:

```
string str, str1;  
str = "ABCDEF";  
str1 = str.substr(2,4);
```

This example assigns string *str1* the value "CDE".

3.4.4 String Class Methods for Matching Patterns

Vera provides several class methods used to match patterns within strings.

search()

The search function searches for a pattern in the string and returns the integer index to the beginning of the pattern. The syntax is:

```
string_name.search(pattern);
```

pattern - The *pattern* parameter must be a string.

The function returns the index value at the start of the string pattern. If the pattern is not found, the function returns -1.

For example:

```
integer i;
string str = "This is a test";
i = str.search("his");
```

This example assigns the index 1 to integer *i*.

match()

The match() function processes a regular expression pattern match. The syntax is:

```
string_name.match(pattern);
```

pattern - The *pattern* parameter must be a string.

The function returns a 1 if the expression is found, and a -1 if the expression is not found. If there is a syntax error in the regular expression, the function returns 0 and sets the status to STR_ERR_REGEX_SYNTAX.

For example:

```
integer i;
string str;
str = "1234 is a number.";
i = str.match("is");
```

This example assigns the value 1 to integer *i* because the pattern "is" exists within string *str*.

The following functions (prematch, postmatch, thismatch, and backref) are used to access the match strings.

prematch()

The prematch() function returns the string before a match, based on the result of the last match() function call. The syntax is:

```
string_name.prematch();
```

For example:

```
integer i;
string str, str1;
str = "1234 is a number.";
i = str.match("is");
str1 = str.prematch();
```

This example assigns the value "1234 " to string *str1*.

postmatch()

The **postmatch()** function returns the string after a match, based on the result of the last **match()** function call. The syntax is:

```
string_name.postmatch();
```

For example:

```
integer i;
string str, str1;
str = "1234 is a number.";
i = str.match("is");
str1 = str.postmatch();
```

This example assigns the value " a number." to string *str1*.

thismatch()

The **thismatch()** function returns the matched string, based on the result of the last **match()** function call. The syntax is:

```
string_name.thismatch();
```

For example:

```
integer i;
string str, str1;
str = "1234 is a number.";
i = str.match("is");
str1 = str.thismatch();
```

This example assigns the value "is" to string *str1*.

backref()

The **backref()** function returns matched patterns, based on the last **match()** function call. The syntax is:

```
string_name.backref(index);
```

index - The *index* is the integer number of the Perl expression being matched. Indexing starts at 0.

This function matches a string with Perl expressions specified in a second string. For example:

```
integer i;
string str, patt, str1, str2;
str = "1234 is a number."
patt = "([0-9]+) ([a-zA-Z .]+)";
i = str.match(patt);
str1 = str.backref(0);
str2 = str.backref(1);
```

This example checks the Perl expressions given by string *patt* with string *str*. It assigns the value "1234" to string *str1* because of the match to the expression "[0-9]+". It assigns the value "is a number." to string *str2* because of the match to the expression "[a-zA-Z .]+". Any number of additional Perl expressions can be listed in the *patt* definition, and then called using sequential index numbers with the *backref()* function.

3.4.5 String Class Methods for Type Conversion

Vera provides several class methods used for type conversion.

atoi()

The *atoi()* function handles a string as an ascii number and converts it to an integer. The syntax is:

```
string_name.atoi();
```

The string must be an ascii number. Underscores are ignored. Spaces and all characters other than digits are invalid. If the ascii string is not a number representation, the function returns 0.

For example:

```
integer i;
string str;
str = "123";
i = str.atoi();
```

This example converts the ascii text "123" and assigns the value 123 to integer *i*.

itoa()

The *itoa()* function converts an integer to an ascii number in a string. The syntax is:

```
string_name.itoa(i);
```

i - The parameter *i* must be an integer. Underscores are ignored. All other characters are invalid.

For example:

```
string str;  
str.itoa(456);
```

This example converts the numeric string value of 456 to ascii text and assigns the value "456" to string *str*.

atohex()

The `atohex()` function handles a string as an ascii hexadecimal number and converts it to a bit value. The syntax is:

```
string_name.atohex();
```

For example:

```
bit [127:0] b;  
string str;  
str = "h12";  
b = str.atohex();
```

This example converts the ascii text value "h12" to the hexadecimal number 'h12 and assigns it to bit *b*.

atooct()

The `atooct()` function handles a string as an ascii octal number and converts it to a bit value. The syntax is:

```
string_name.atooct();
```

For example:

```
bit [127:0] b;  
string str;  
str = "o12";  
b = str.atooct();
```

This example converts the ascii text value "o12" to the octal number 'o12 and assigns it to bit *b*.

atobin()

The `atobin()` function handles a string as an ascii binary number and converts it to a bit value. The syntax is:

```
string_name.atobin();
```

For example:

```

bit [127:0] b;
string str;
str = "b01";
b = str.atobin();

```

This example converts the ascii text value "b01" to the binary number 'b01 and assigns it to bit *b*.

bittostr()

The `bittostr()` function converts the text in bits to text in strings. The syntax is:

```
string_name.bittostr(bit[high:low] bit_string);
```

For example:

```

bit [127:0] b;
string str;
b = "Hello";
str.bittostr(b);

```

This example converts the bit string "Hello" to a string and assigns the value "Hello" to string *str*.

3.4.6 Additional System Functions

The following system functions are not part of the string class methods. They are independent functions that manipulate strings or are related to strings.

sprintf()

The `sprintf()` system task sends output to a string variable. The syntax is:

```
sprintf(string_name, string_format);
```

The output specified by *string_format* is assigned to the string *string_name*.

For example:

```

string S0;
string str;
S0 = "S0 string";
sprintf(str, "S0 is %s\n", S0);

```

This example assigns the string "S0 is SO string" to string *str*.

sscanf()

The `sscanf()` system task reads input from a string. The syntax is:

```
sscanf(string_name, string_format);
```

The input specified by *string_format* is assigned to the string *string_name*.

For example:

```
string s1;
int i1, i2, i3;
s1 = " 123 'h10 4567";
sscanf(s1, " %d %h %2d", i1, i2, i3);
```

This example assigns the values 123, 16, and 45 to the variables *i1*, *i2*, and *i3* respectively.

3.5 Enumerated Types

Enumerated types are a user-defined list of named integer constants. As discussed in Section 3.2.6, the syntax to declare an enumerated type is:

```
enum category = list;
```

So, for example, we could have:

```
enum colors = red, green, blue, yellow, white, black;
```

This operation assigns a unique number to each of the color identifiers and allows us to create a new data type of type "colors."

```
colors new_color;
integer val;

new_color = green;
new_color = 1; // Invalid assignment.
```

This example assigns the color green to the *colors* variable *new_color*. The second assignment is invalid because of the strict typing rules used by enumerated types.

Elements within enumerated type definitions are assigned identifiers, which are numbered consecutively, starting from 0. In our example, red is assigned 0, green is assigned 1, and so on.

You can further specify identifiers in the element list in several ways:

- **name** This associates the next consecutive integer with *name*.
- **name[N]** This generates N names in the sequence (name0, name1, ..., nameN-1) where N must be a constant integer.
- **name[n:m]** This creates a sequence of names starting with *namen* and counting up (or down) to *namem*.
- **name=N** This assigns the constant N to *name*.

For example:

```
enum instructions = add=10, sub[5], jmp[6:8];
```

This example assigns the number 10 to the enumerated type *add*. It also creates the enumerated types *sub0*, *sub1*, *sub2*, *sub3*, and *sub4*, and assigns them the values 0-4 respectively. Finally, the example also creates the enumerated types *jmp6*, *jmp7*, and *jmp8*, and assigns them the values 6-8 respectively.

3.5.1 Enumerated Types in Numerical Expressions

Elements of an enumerated type or an enumerated type variable can be used in numerical expressions. The value used in the expression is the numerical value assigned to the enumerated type element. For example:

```
colors new_color;
integer val1, val2;

val1 = blue * 3;
new_color = yellow;
val2 = new_color + green;
```

From our previous declaration, *blue* has a numerical identifier of 2. This example assigns *val1* a value of 6 (2*3). This example then assigns *val2* a value of 5 (2+3).

Note – Assignments to enumerated type variables are strongly typed. Thus, assigning numerical expressions to enumerated type variables causes compilation errors.

3.5.2 Increment and Decrement Operations on Enumerated Types

The operators *++*, *--*, *+=*, and *-=* have special meanings on enumerated type variables.

<i>enum_var++</i>	Assigns the next member (as defined by the definition order) to <i>enum_var</i> . The first member is selected if <i>enum_var</i> is currently holding the last member.
<i>enum_var--</i>	Assigns the previous member (as defined by the definition order) to <i>enum_var</i> . The last member is selected if <i>enum_var</i> is currently holding the first member.
<i>enum_var += val</i>	Assigns the <i>val</i> -th next member to <i>enum_var</i> . A wrap to the beginning of the list occurs when the end of the list is reached.
<i>enum_var -= val</i>	Assigns the <i>val</i> -th previous member to <i>enum_var</i> . A wrap to the end of the list occurs when the beginning of the list is reached.

Note – “*enum_var* += *val*,” is different than “*enum_var* = *enum_var* + *val*,” The former is legal while the latter is illegal because “*enum_var* + *val*” is evaluated to a numerical expression which, in turn, cannot be assigned to an enumerated type variable.

3.6 Vera Operators

Vera uses a set of standard operators for expressions and concatenation.

3.6.1 Operators

Table 3-8 lists the basic Vera operators.

Table 3-8 Vera Operators

Operator	Semantics
{ }	concatenation
{ (}	concatenation left of assignment
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
==?	wild equality
!=?	wild inequality
~	bit wise negation
&	bit wise and
&~	bit wise nand
	bit wise or
~	bit wise nor
^	bit wise exclusive or

Operator	Semantics (<i>Continued</i>)
<code>^~</code>	bit wise exclusive nor
<code>&</code>	unary and
<code>~&</code>	unary nand
<code> </code>	unary or
<code>~ </code>	unary nor
<code>^</code>	unary exclusive or
<code>~^</code>	unary exclusive nor
<code><<</code>	left shift
<code>>></code>	right shift
<code>?:</code>	conditional

The conditional operator (`?:`) and the comparison operators `==`, `!=`, `===`, and `!==` can be used to compare variables of the same type for all base types.

The wild equality operator (`==?`) treats an *x* value or *z* value in a given bit position (for bit values) as a wildcard. It matches any bit value (0, 1, z, or x) in the value of the expression being compared against it. Consider the following vector and some comparisons against it:

```
a = 8'b00110101
(a ==? 8'b00110101) is true
(a ==? 8'b00010101) is false
(a ==? 8'b00x10101) is true
```

The same rule applies to the wild inequality operator (`!=?`).

The precedence order of binary operators is defined in Table 3-9.

Table 3-9 Precedence Order of Binary Operators

Operator	Precedence
<code>! ~</code>	Highest precedence
<code>* / %</code>	
<code>+ -</code>	
<code><< >></code>	
<code>< <= > >=</code>	
<code>== != === !== ==? !=?</code>	
<code>& &~</code>	
<code>^ ^~</code>	

Operator	Precedence
~	
&&	
?:	Lowest precedence

3.6.2 Concatenation

The syntax for concatenation is:

```
variable = {var1, var2, ..., varN};
```

Variable is the result of the concatenation. The arguments are concatenated sequentially.

For example:

```
bit [6:0] data;
bit parity;
bit [7:0] foo;
foo = {data, parity};
```

Multiple concatenation is also supported. For example:

```
{ 32 {1'b1 } }
{ 4 {foo, moo} }
```

This example concatenates *foo* and *moo* four times.

Vera uses the left brace to open a block and also for concatenation. This creates a conflict when using the left brace for concatenation on the left-hand side of assignments. Therefore, Vera uses a single quote to prefix the left brace when it is used for concatenation on the left. For example:

```
'{data,packet,parity} = 256'b0;
```

3.7 Variable Assignment

The variable assignment is the primitive operation to set a value for a variable. The syntax to assign values to variables is:

```
variable_name operator assign_expression
```

For example:

```

i = 0;
a = 1'b0
temp[3:0] = 4'b1000;
memory [53] = 8'b0x0x0x0x;
b_port = portid ? port0 : port1;

```

There are two types of assign operators: the = operator is called the simple assignment operator; all others are called compound assignment operators.

For the compound assignment operator, the expression `a <operator>= b` is equivalent to `a = a <operator> b`.

For example, the following are equivalent:

```

i = i + 5;
i += 5;

```

Vera supports the C-style ++ and -- operators. For example:

```

result = 5;
a = result++;
a = ++result;

```

The second line accesses the variable and then increments it. The third line increments the variable and then accesses it.

Vera does not support assignment recursion. The following is an illegal assignment:

```

a = b = c;

```

The Vera compiler does type checking. Bind_var variables can be used in simple assignments through the assignment operator, can be used in conditional assignments (?:), or can be passed as arguments. Event variables cannot be used in assignments because they are used only for triggering purposes. All other combinations of integer and bit variables are valid.

Note – To help avoid mistakes, <assignments> are not expressions. Hence, statements like the following are invalid:

```

if(a=b) a=c+d; // should be a==b

while(a=b) a=random(); // should be a==b

```

4. Basic Vera Programming

This chapter documents the basic elements of Vera programming. It details the fundamental program structure used in all Vera programs, introduces signal interfaces, and describes several of the elementary signal operations. This chapter includes these sections:

- Vera Programming Overview
- Program Structure
- Signal Declarations
- Signal Operation
- Asynchronous Operations
- Subroutines

4.1 Vera Programming Overview

Vera programming involves the integration of several key components of a Vera testbench: the main Vera program, the Vera-HDL interface, the clocking mechanisms, signal operations, and Vera subroutines.

The main Vera program is where global variables are defined, executable statements are carried out, and calls to subroutines are made. The main Vera program is the centerpiece of the Vera testbench.

The Vera-HDL interface defines how Vera interacts with the hardware description created in either Verilog or VHDL. It establishes how the HDL signals are driven and sampled by Vera.

The clocking mechanisms determine the timing of the HDL signals. This directly affects signal driving and sampling. The clocks also determine timing for Vera processes, which affects synchronization of concurrent threads.

The signal operations include the driving and sampling primitives in the Vera language. They handle all sampling and driving of signals. Other fundamental signal operators include the expect primitive and synchronization.

Vera can call subroutines (tasks and functions) from the main Vera program and from within subroutines. These subroutines include pre-defined tasks and functions as well as user-defined tasks and functions.

4.2 Program Structure

Vera programs have a single, top-level structure denoted by the keyword **program**. The syntax is:

```
#include <vera_defines.vrh>
#include "filename"
#define text_macro


```

The top block contains tasks, procedures, and function definitions. Vera, like C, supports two levels of hierarchy: top (containing the **program** statement) and others. Variables defined in the top block are global and are seen in any task/procedure/function anywhere in the entire set of testbench files (unless the same name is used to define a local variable). Using same named variables is called "shadowing" and should be used with caution because of variable conflicts that can arise.

The syntax of a complete Vera testbench is flexible. It has one program and any number of submodules in any order. Forward references are allowed in any file. However, references to tasks or functions in other files require external references to be declared. The compiler accepts task and function declarations before and after the top program.

Submodules, declarations, and external declarations are discussed in more detail in subsequent chapters. Note, however, that they can be used in any order.

File inclusion, text macros, and preprocessor commands are discussed in Section 18.2, "Preprocessor." Coverage blocks are discussed in Chapter 11., "Functional Coverage." Classes are discussed in Section 8.1, "Classes and Objects." HDL and C/C++ subroutines are discussed in Chapter 14, "Vera-HDL Task Calls" and Chapter 15, "Calling C/C++ Functions."

The other major components of the main Vera program are discussed in detail in subsequent sections in this chapter.

4.3 Signal Declarations

Vera signal declarations determine how Vera testbenches are connected to HDL designs. There are three methods of signal declaration: Vera-HDL interface specifications, virtual port definitions, and signal binds.

4.3.1 Vera-HDL Interface Specifications

The Vera-HDL interface identifies a set of signals through which Vera communicates with the HDL environment. The interface also determines how Vera drives and samples those signals. Finally, the interface defines the clock to which the signals are tied. The syntax for interface specifications is:

```
interface interface_name {
    signal_direction signal_width signal_name signal_type skew;
}
```

signal_direction - The *signal_direction* specifies the direction of the signal with respect to Vera.

It must be **input**, which specifies signals that go from the HDL to Vera, **output**, which specifies signals that go from Vera to the HDL, or **inout**, which specifies bidirectional signals.

signal_width - The *signal_width* is a bit vector specifying the width of the signal. It must be in the form *[high:0]*.

signal_name - The *signal_name* identifies the signal being defined. It is the top-level name of the HDL signal being connected.

signal_type - The valid signal types and their definitions are listed in Table 4-1.

Table 4-1 Signal types

Signal Type	Operation
NHOLD	Output is driven on the negative edge of the interface clock.
PHOLD	Output is driven on the positive edge of the interface clock.
NR <i>value</i>	Output is driven on the falling edge of the interface clock for 1 cycle. Then it returns to <i>value</i> , which can be 0, 1, X, or Z.
PR <i>value</i>	Output is driven on the rising edge of the interface clock for 1 cycle. Then it returns to <i>value</i> , which can be 0, 1, X, or Z.
NSAMPLE	Input is sampled (evaluated) at the negative edge of the interface clock.
PSAMPLE	Input is sampled (evaluated) at the positive edge of the interface clock.
CLOCK	Specifies the clock, which qualifies other interface signals.

For one-direction signals, only one signal type can be used. Input signals are sampled, and output signals are driven. Multiple signal types should be associated with bidirectional signals.

skew - The *skew* determines how long before or after the clock edge the signal is driven or sampled. The skew must be in this format:

#value

All drives are done at or after the appropriate clock edge (positive skew). This means that drives can only take positive skew values. All sampling is done at or before the appropriate clock edge (negative skew). This means that samples can only take negative skew values. This limitation so that causality is not violated; it is impossible to sample a signal, use the value to determine a drive, and then drive a signal on a single clock edge.

Note – To avoid race conditions, you should use a skew of at least -1 for all signal sampling.

Figure 4-1 shows an example of positive and negative skews.

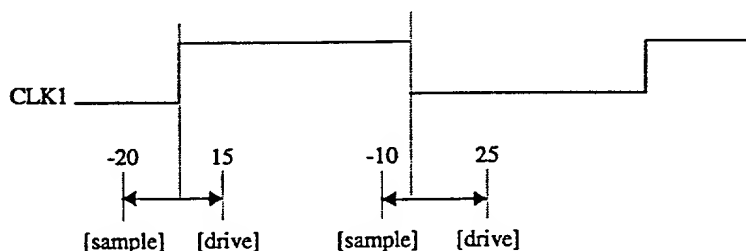


Figure 4-1 Example of positive and negative skews

This is an example interface declaration:

```
interface arb {
  output [1:0] request PHOLD #1;
  input [1:0] grant PSAMPLE #-1;
  inout [7:0] data PSAMPLE #-1 PHOLD #1;
}
```

4.3.1.1 HDL Nodes

HDL nodes allow you to bind a Vera port directly with a wire or port in an HDL design. The syntax to declare an HDL node is:

signal_direction signal_name signal_type skew node_method "signal_path";

node_method - The *node_method* specifies the type of HDL design you are linking to. It must be either *verilog_node*, which links to Verilog designs, or *vhdl_node*, which links to VHDL designs.

signal_path - The *signal_path* is the HDL path to the specified signal. It must be surrounded by double quotes.

Note – VHDL nodes do not support signal subfielding.

For example:

```
inout [31:0] D1 PSAMPLE verilog_node "sys.cpu2.p0_D1";
output req PSAMPLE vhd1_node "/sys/arb/p0_strb";
```

Note the preceding slash and the use of slashes in the `vhd1_node` construct to specify the hierarchal path.

When HDL nodes are used, the interface signals are declared as internal wires in the Vera-HDL shell (*vera_shell*) and are directly connected to the specified HDL node. If all the signals are declared using the node construct, the *vera_shell* module has no input or output signals. It can then be included as a top module with the HDL design. As long as the interface does not change, the resulting *vera_shell*-HDL module can be used with different test benches without redeclaring the signals.

4.3.1.2 Clocking Domains

Each interface defines a clock domain. The syntax to define a clock domain within the interface specification is:

```
input clock_name CLOCK;
```

clock_name - The *clock_name* is the name of the HDL clock driving the simulation.

The clock domain specifies when clock edges occur and determines when signals are driven and sampled.

For a given interface specification, a clock domain establishes a relationship between the clock and when the other given interface signals are driven or sampled.

Clock domains can be overlapped. The same signal can be associated with multiple clocks via multiple interface specifications. However, despite multiple interfaces, it is still a single signal and cannot be driven to two values at the same time.

4.3.1.3 Signal Depth

Input signals are typically signalled in the current cycle. However, input signals can be sampled in previous cycles. To reference a signal in a previous cycle, the signal depth must be specified in the signal declaration. The syntax to specify the signal depth is:

```
signal_direction signal_name signal_type skew depth value;
```

value - The *value* specifies the number of cycles that are stored for back-reference. If you want to reference a signal 3 cycles back, this value must be at least 3. The value must be an integer.

For example:

```
inout [7:0] data PSAMPLE PHOLD #1 depth 5;
```

The syntax to reference a signal in a previous cycle is:

```
interface_name.signal_name.N
```

interface_name - The *interface_name* is the name of the interface in which the signal is defined.

signal_name - The *signal_name* is the name of the signal being referenced.

N - *N* specifies how many previous cycles to look back when evaluating the signal. The current cycle is 0, the previous cycle is 1, and so on. If no signal depth is specified, the default is the current cycle.

4.3.2 Virtual Ports

Virtual ports are sets of port signal names grouped together under a given name. Ports are defined outside of the main program block. The syntax to define a virtual port is:

```
port port_name {port_signal1; port_signal2; ...; port_signalN;}
```

port_name - The *port_name* is the name of the virtual port you are creating.

port_signalN - *Port_signalN* is a generic signal name that is assigned a specific HDL signal through a bind declaration. Multiple port signal names are separated by semi-colons (;).

Virtual ports define a set of signal placeholders through generic signal names. The signal names must then be bound to the actual signals in the interface using a bind declaration.

This is an example port declaration:

```
port rcv_port {frame_n;valid_n;busy;packet;}
```

4.3.3 Bind

Binds tie virtual ports to actual interface signals. Binds are declared outside of the main program block. The syntax to declare a bind is:

```
bind port_name bind_name {
    port_signalN interface_name.signal_name;
}
```

port_name - The *port_name* is the virtual port name you want to bind.

bind_name - The *bind_name* is the name of the bind you are creating.

port_signal_name - The *port_signalN* is the name of the generic signal names that you are including in the bind. Generally, all of the signals in the port are bound. However, you can bind only selected signals if you want.

interface_name - The *interface_name* is the name of the interface to which you are binding the port.

signal_name - The *signal_name* is the name of the signal you are binding to the port. You can specify signal subfields using *signal_name[x:y]*.

Each bind in Vera is unique, and Vera assigns it a numeric ID. The bind can then be referred to by the name of the bind or by the numeric ID (obtained using the `get_bind_id()` system function).

When using subfield binding, you can specify further subfields. For example:

```
bind prt mybind{
  a intf.sign[7:5];
}
```

This example assigns a 3bit subfield to the port signal *a*. You can specify a subfield within that field as well:

```
mybind.$a[2:1]
```

This signal refers to the subfield in signal *a*, which corresponds to `intf.sign[7:6]`.

This is an example bind using the port declaration from the previous section:

```
bind rcv_port iport0{
  frame_n router.frame_n0;
  valid_n router.valid_n0;
  busy router.busy_n0;
  packet router.di0;
}
```

4.3.3.1 Bind Functions and Tasks

Vera provides several system tasks and functions used with binds.

`get_bind()`

The `get_bind()` system function returns the current binding in a task or function. The syntax is:

```
get_bind([ID]);
```

ID - The *ID* can be either a numeric ID (obtained using the `get_bind_id()` system function), or it can be the string name of the bind.

The `get_bind_id()` system function returns the bind with the specified ID. If no argument is passed, the bind in the current system task is returned. If no match is found, a NULL value is returned.

For example:

```
task handshake (bind_var iport, bit direction, bit my_data)
{
  drive_data(get_bind(), direction, my_data);
}
```

This example declares the task `handshake`, which is passed three arguments when it is called, including a bind. The task then calls the function `drive_data` and passes the current bind and two other arguments when it is called. This ensures that the function `drive_data` operates on the same port that `handshake` uses.

`get_bind_id()`

The `get_bind_id()` system function returns the unique ID number for a given bind. The syntax is:

```
get_bind_id(bind_expression);
```

bind_expression - The *bind_expression* can be a bind name, a bind variable, a port variable, or any expression that returns a bind (such as `get_bind()`). If it is not specified, the current bind ID is returned.

For example:

```
task handshake (bind_var iport, bit direction, bit my_data)
{
    x = get_bind_id();
    drive_data(get_bind(x), direction, my_data);
}
```

This example gets the current bind ID and passes it to the `get_bind()` system function. Note that the variable `x` must be declared as a `bind_var`.

4.3.4 Using Ports and Binds

The power of ports and binds comes from the ability to pass them as arguments to tasks and functions so that the tasks and functions work on the desired signals. This allows you to define generic tasks and functions, independent of the signals they act on.

The use of ports and binds involves *bind_var* and *port* variables.

4.3.4.1 `bind_var` Variables

`Bind_var` variables store binds. The syntax to declare a `bind_var` is:

```
bind_var variable_name = initial_value;
```

Specific binds are assigned to `bind_vars`, and the `bind_vars` are passed as arguments to tasks and functions to specify which signals the tasks and functions operate on. For example:

```
port myport(mysignal1;mysignal2;mysignal3);
```

```

bind myport mybind {
    mysignal1 myinterface.signal1;
    mysignal2 myinterface.signal2;
    mysignal3 myinterface.signal3;
}

bind_var mybindvar;

mybindvar = mybind;

function integer myfunction (bind_var testport) {
    ...
}

```

This code creates the virtual port `myport` and binds it to the corresponding signals declared in `myinterface`. Then it declares the `bind_var mybindvar`. The code also defines the generic function `myfunction` with the arguments `testport1`. This line, within the main program module calls the function with the appropriate port:

```
myfunction(mybindvar);
```

This function call passes the `bind_var mybindvar` to the function so that the function operates on the desired signals.

`Bind_vars` can be declared as arrays just as any other data type.

`Bind_vars` can be assigned different binds, just like an integer variable can be assigned different values.

Note – You must be careful when assigning new binds to `bind_vars` and passing them as arguments because any bind can be passed to any `bind_var`. For example, if a function acts on 3 signals and you pass a bind with only 2 signals, a runtime error will occur. There is no type checking to ensure that the right bind is being passed.

4.3.4.2 Port Variables

Port variables store binds. The syntax to declare a port variable is:

```
port_name port_variable = initial_value;
```

port_name - The `port_name` is the name of the port data type you are referencing.

port_variable - The `port_variable` is the name of the port variable you are declaring.

initial_value - The `initial_value` can be any existing port. If it is not set, the `port_variable` has a NULL value until it is assigned a port.

When a virtual port is declared, that virtual port becomes a new data type (much like an enumerated type). Variables of that type can then be declared and passed as arguments to generic functions and tasks much in the same way that `bind_vars` are.

The key difference between port variables and `bind_vars` is that only binds of the port data type from which the port variable was declared can be passed to the port variable. For example:

```
port myport1(mysignal1;mysignal2;mysignal3);
port myport2(mysignalA;mysignalB;mysignalC);

bind myport1 mybind1 {
    mysignal1 myinterface.signal1;
    mysignal2 myinterface.signal2;
    mysignal3 myinterface.signal3;
}

bind myport2 mybind2 {
    mysignalA myinterface.signal1;
    mysignalB myinterface.signal2;
    mysignalC myinterface.signal3;
}

myport1 myportvar = mybind1;
```

This code block creates the virtual ports `myport1` and `myport2`, and binds the signals to the corresponding signals declared in `myinterface`. Then it declares the port variable `myportvar` of type `myport1` with the initial bind `mybind1`.

A `bind_var` could be assigned the bind `mybind2`. However, assigning the bind `mybind2` to `myportvar` results in an error because the port types are not the same.

Port variables can be declared as arrays just as any other data type.

Note – Port variables provide a stricter method of type checking not afforded by `bind_vars`. For this reason, port variables are generally a preferred means of storing binds.

4.3.4.3 Direct Bind Signals

Once a port and bind have been declared, bind signals can be referenced directly. The syntax to reference a bind signal is:

```
bind_or_port_name.signal_name;
```

bind_or_port_name - The *bind_or_port_name* is the name of the bind in which the signal is referenced or the name of the port variable that includes the bind.

signal_name - The *signal_name* is the name of the signal as defined in the bind specified (either the direct bind specified or the bind with which the port variable is associated).

For example:

```
port rcv_port {frame_n, packet}
```

```

bind rcv_port iport0(
    frame_n router.frame_n0;
    packet router.di0;
)

rcv_port port_val = iport0;
rcv_port ports[3];
ports[1] = iport0;

router.frame_n0 = 0;
port_val.$frame_n = 0;
ports[1].$frame_n = 0;

```

This example declares the virtual port `rcv_port` and creates the bind `iport0`. Then it declares the port variables `port_val` and `ports[3]`, and assigns the bind `iport0` to `port_val` and `ports[1]`. The last 3 lines are all equivalent methods of referencing the same signal.

4.3.5 Dynamic Binding

Vera's dynamic binding capabilities allow you to connect to an HDL signal at runtime and change specified interfaces during the simulation based on external conditions. The dynamic binding capabilities include void binds, runtime bind declaration, and runtime signal mapping.

4.3.5.1 Void Binds

Vera allows you to leave port signals unassigned using the `void` construct. The `void` declaration is made within the bind:

```
port_signal_name void;
```

Using the `void` construct creates a port signal that has no interface signal assigned to it. Interface signals can be assigned at runtime during the simulation. This is an example of a port and bind declaration using void binds:

```

port myport1
{
    mysignal1;
    mysignal2;
}

bind myport1 mybind1 {
    mysignal1 myinterface.signal1;
    mysignal2 void;
}

```

4.3.5.2 Runtime Bind Declaration

Vera allows you to declare new binds at runtime by instantiating port variables. The syntax for runtime bind declarations is:

```
port_name = new;
port_name = new bind_name;
```

port_name - The *port_name* is the name of the port variable being instantiated.

bind_name - The *bind_name* optionally specifies a bind of the same port type as the port variable, which specifies how the port signals are initially bound.

The first construct instantiates the port variable. The port variable is local to the function it is instantiated in, or global if it is instantiated in the main program. The new port variable is of the same bind type as it was declared with all of the port signals unassigned. For example:

```
task my_task()
{
    Myport port1, port2;

    port1=new;
}
```

This example instantiates the port variable *port1*, which will be of bind type *Myport* with each signal being assigned a *void* value. The resultant bind is:

```
bind Myport port1
{
    signal1 void;
    signal2 void;
    ...
    signalN void;
}
```

signalN - The signals are the port signal names specified in the original bind declaration *Myport*.

Alternatively, you can optionally specify another bind of the same port type as the port variable:

```
port_name = new bind_name; -
```

This construct instantiates the port variable as a bind of the same type as the specified bind.

Note - The specified bind and port variable must be of the same port type.

For example:

```
bind Myport Mybind
{
    signal1 value1;
    signal2 value2;
```



```

...
    signalN valueN;
}

task my_task()
{
    Myport port1, port2;

    port1=new Mybind;
}

```

This example instantiates the port variable *port1*, which will be of bind type *Mybind*. The resultant bind is:

```

bind Myport port1
{
    signal1 value1;
    signal2 value2;
    ...
    signalN valueN;
}

```

signalN - The *signals* are the signals specified in the original bind declaration *Mybind*.

valueN - The *signal values* are the values as specified in the bind *Mybind*.

4.3.5.3 Runtime Signal Mapping

Vera allows you to map port signals to HDL signals at runtime using the system function *signal_connect()*. The syntax is:

```

signal_connect(port_signal, target_signal [, attributes [, clock] ]);

```

port_signal - The *port_signal* is the signal being mapped. It must be a port signal of an instantiated bind.

target_signal - The *target_signal* is the new signal to which the *port_signal* is mapped. It can be a signal reference or a string specifying an HDL node. You can specify subfields within the *target_signal*.

attributes - The *attributes* optionally specify how the new connection is made. Multiple attributes can be assigned in any order. The attributes with possible values are listed in Table 4-2.

Table 4-2 `signal_connect()` attributes and values

Attribute	Possible Values
<code>dir</code>	<code>input</code> , <code>output</code> , or <code>inout</code>
<code>width</code>	integer width of signal to connect (default is signal width on HDL side)
<code>itype</code>	<code>NSAMPLE</code> or <code>PSAMPLE</code>
<code>otype</code>	<code>PHOLD</code> , <code>NHOLD</code> , <code>PDRIVE</code> , <code>NDRIVE</code> , <code>NR0</code> , <code>NR1</code> , <code>NRX</code> , <code>NRZ</code> , <code>PR0</code> , <code>PR1</code> , <code>PRX</code> , or <code>PRZ</code>
<code>depth</code>	0 or 1 (default is 0)
<code>iskew</code>	any non-positive integer (default is 0)
<code>oskew</code>	any non-negative integer (default is 0)

This is an example string declaring signal attributes:

```
"dir=input width=1 itype=PSAMPLE iskew=-3"
```

Note that the string is encapsulated in double quotes (" "). The string is not case sensitive.

If the *target_signal* is an HDL node, the attributes must specify at least a minimum complete set (direction and signal type). If the *target_signal* is a specific signal, the attributes simply modify the existing set of attributes for that signal.

clock - The *clock* specifies the clock for any generated interfaces. It can be a reference to an interface signal, or it can be a string specifying a signal on the HDL side. Regardless, it must be a 1-bit signal (subfields are not allowed). If it is not specified, `SystemClock` is used.

When the `signal_connect()` system function is called, the port signal is connected to the target signal as specified. If the port signal was previously connected, that connection is lost when the call is made. The connection that is made when the function is called depends on how the call is made.

Only *port_signal* and *target_signal* are specified

If only the *port_signal* and *target_signal* are specified, the *port_signal* is simply remapped to the *target_signal*. The *target_signal* must be a signal reference. For example:

```
signal_connect(port1.$b, dff.q);
signal_connect(port1.$b, port1.$a);
signal_connect(port1.$b $a);
```

The first example connects signal b from port1 to signal dff.q. The second example connects signal b from port1 to signal a from port1. The third example connects signal b from port1 to signal a from the port with which the task (with port) is called.

***target_signal* specifies an HDL node**

If the *target_signal* is a string specifying an HDL node, you **MUST** include the *attributes*. When the function call is made with the *target_signal* specifying an HDL node, a new interface is created with the given attributes. The clock for the interface is either the clock specified in the function call or `SystemClock` if no clock is specified. All the signal interfaces are force driven by Vera at runtime. This drive overrides any HDL driving or static interface connections.

Note – Using this configuration, you cannot connect multi-bit registers or portions of registers as `output` or `inout` signals.

For example:

```
signal_connect(port1.$b, "test_top.dff.q[3:0]", "dir=input
width=4 itype=PSAMPLE iskew=-3", dff.clk);
```

This example creates a new interface with a single connection from the signal `port1.$b` to the HDL node specified by `"test_top.dff.q[3:0]"`. On the DUT side, the signal has a width of 8 bits, but in this example only the 4 least significant bits are connected. The signal has the specified attributes. The interface clock is `dff.clk`.

Note – When using this `signal_connect()` configuration with Verilog-XL, you must use the `-x` switch at runtime. When using this configuration with VCS, you must compile the code with `-P $VERA_HOME/lib/vcs_pli_dyn.tab`.

***target_signal* specifies a signal and *attributes* or *clock* is specified**

If the *target_signal* specifies a signal or interface and either the *attributes* or *clock* (or both) is specified, a new interface is created. The signal is connected to the specified target and the new interface has the given attributes and clock. This configuration for the `signal_connect()` call is useful for sampling the same signal on different clock edges or with a different clock. By changing the attributes or assigning a new clock to the signal, you can sample the signal under varying conditions. If you need to sample the same signal with a different clock without changing any attributes, you can specify the attributes with an empty string("").

Note – When using this `signal_connect()` configuration with Verilog-XL, you must use the `-x` switch at runtime. When using this configuration with VCS, you must compile the code with `-P $VERA_HOME/lib/vcs_pli_dyn.tab`.

4.3.5.4 Dynamic Binding Example

This is an example of how dynamic binding is used:

```

port port_channel
{
    bus_a;
    bus_b;
} // a port with just 2 bits

task create_bindings
(
    • var port_channel arr_binds[], // destination assoc. array of ports
    integer how_many_ports, // how many ports
    string sig_options, // signal options, e.g. "dir=input width=1 ..."
    string sig_clock // the clock to connect to, e.g. "main_mod.clk"
)
{
    integer i;
    string verpath;

    for (i = 0; i < how_many_ports; i++)
    {
        arr_binds[i] = new; // create a binding for this array index
        sprintf(verpath, "main_mod.bus_a[%0d]", i); // generate path for bus_a
        signal_connect(arr_binds[i].$bus_a, verpath, sig_options, sig_clock);
        sprintf(verpath, "main_mod.bus_b[%0d]", i); // generate path for bus_b
        signal_connect(arr_binds[i].$bus_b, verpath, sig_options, sig_clock);
    }
}

```

This example creates an associative array of bindings of type `port_channel`, and assigns to them each bit of `main_mod.bus_a` and `main_mod.bus_b`. Each element of the associative array is assigned a different bit of `bus_a` and `bus_b`. Note the use of string expressions to specify the HDL path.

4.4 Signal Operation

Vera separates its timing and signal operations, which makes test development and maintenance easier to handle. The timing is defined in the interface specification, which simplifies signal operation in that the skew and clock are predefined.

Vera provides four primitive statements that operate on interface signals:

- Synchronization
- Drive
- Sample
- Expect.

Vera also provides a value change alert (VCA) for use with signals.

4.4.1 Synchronization

The synchronization operator (@) is used to perform explicit synchronization. The syntax is:

```
@ (clock_edge interface_signal);
```

clock_edge - The *clock_edge* optionally specifies the clock edge at which the synchronization occurs, as defined in the interface declaration. It must be either **negedge**, which specifies a negative or falling clock edge, or **posedge**, which specifies a positive or rising clock edge. If no clock edge is specified, the synchronization occurs on the next change in the specified signal.

interface_signal - The *interface_signal* specifies the signal to which the synchronization is linked. It can be any defined signal or **CLOCK**. The interface signal can be any subfield of a signal as well. If a signal is not specified, the interface is clocked by **SystemClock**.

Note - Only 1 bit signals can be synchronized on the **posedge** or **negedge**.

If the interface signal is a subfield of a signal, the synchronization occurs on the first change of the signal subfield. If the subfield is a 1-bit subfield, you can synchronize on clock edges. If you specify variables in the subfield, Vera evaluates the variables at runtime.

You can use the **OR** keyword to specify multiple interface signals. If you specify more than one signal, the synchronization occurs on the next change of any of the listed signals.

These are some example synchronization statements:

```
@( foo_bus.ack_1 );
@( CLOCK );
@( posedge foo_bus.clock );
@( negedge intf1.sign[a:b]);
```

The first example synchronizes to the clock edge corresponding to the next change of the **ack_1** signal, as declared in the **foo_bus** interface specification. The second example synchronizes to the **SystemClock**. The third example synchronizes to the positive edge of the interface clock. The last example synchronizes to the falling edge of the specified subfield.

Note that at initialization, HDLs can create edges at **time = 0** (for example, going from **X** to the initialized value). This means that synchronization conditions can be set before initialization of the signal.

4.4.2 Drive

Drives set the value of output interface signals. The syntax to drive a signal is:

```
delay signal_name range drive_operator expression;
```

delay - The *delay* specifies the number of cycles that pass before the signal is driven. It is in the form **@n** where *n* is the number of cycles.

signal_name - The *signal_name* is the name of the signal being driven.

range - The *range* specifies which bits of the signal are driven. If no *range* is specified, the entire signal is driven.

drive_operator - The *drive_operator* must be either `=`, which specifies a blocking drive, or `<=`, which specifies a non-blocking drive.

expression - The *expression* can be any valid Vera expression.

These are some drive examples:

```
foo_bus.data[3:0] = 4'h5; // blocking drive
@1 foo_bus.data <= 8'hz; // non-blocking drive
```

4.4.2.1 Blocking and Non-Blocking Drives

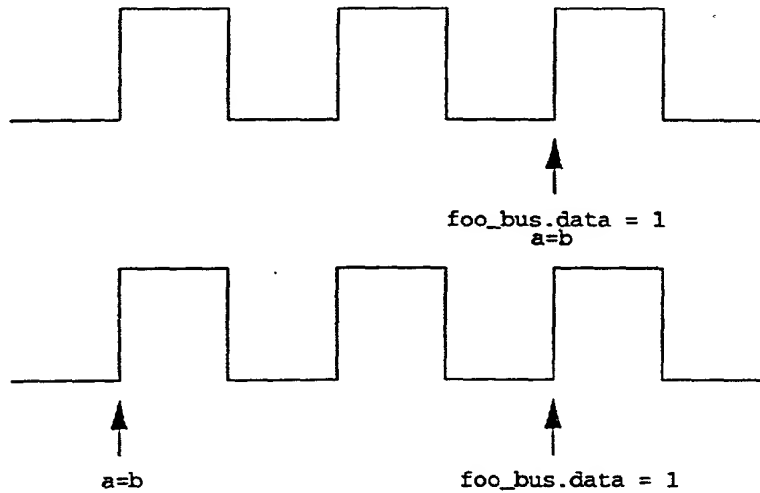
There are two types of drives specified by the drive operator: blocking and non-blocking. Blocking drives suspend Vera execution until the number of HDL cycles specified in the delay passes. Note that the specific time within the cycle that the drive unblocks is determined by the clock edge on which the drive takes place (NHOLD or PHOLD). Once the delay passes, Vera execution resumes. Non-blocking drives schedule the drive at a future cycle and Vera execution continues. When the specified cycle occurs, the drive is executed.

Note – For performance reasons, input and output skews are implemented with delay buffers to minimize the amount of interaction of Vera with the simulation. Unless you use the `delay()` system task or asynchronous signal operations, Vera only interacts with the simulation at clock edges.

These are some blocking and non-blocking examples:

```
@3 foo_bus.data = 1;
a = b;
```

```
@3 foo_bus.data <= 1;
a = b;
```



The first block is a blocking drive. Three cycles must pass before the second line is executed. The second block is a non-blocking drive. The drive is delayed three cycles. The second line is executed, and the simulation advanced three cycles before the first drive is executed.

4.4.2.2 Strong and Soft Drives

There are two strengths of drives: strong (default) and soft. The syntax to declare a soft drive is:

```
delay signal_name range drive_operator expression soft;
```

A given signal should only be driven by a single strong drive at any given time. Multiple strong drives at the same time result in conflicting drives. Conflicting strong drives drive the signal to X and result in a simulation error. However, conflicting soft drives can drive a signal to X without causing a simulation error. Finally, if a signal is driven by a conflicting strong and soft drive, the strong drive dominates and the signal is driven by the strong drive.

4.4.2.3 Void Drives

Drives block until the appropriate driving clock edge occurs. If you want to wait for this clock edge without actually driving the signal, use the void construct. The syntax is:

```
delay signal_name = void;
```

For example:

```
@1 iport0.packet = void;
```

4.4.3 Sample

Samples assign the value of a signal to a variable. The syntax is:

```
variable = signal_name;
```

The signal value can be an interface signal or a port signal, but it must be an **input** or **inout** signal. It is sampled at the next sampling point (specified in the interface file) and the value is assigned to the variable. It cannot be delayed like drives can. Remember that you can sample previous cycles using the signal depth construct (see Section 4.3.1.3, "Signal Depth"). Also remember that you can sample subfields within the signal by specifying a specific subfield in the signal width.

Note – When you sample a signal in an expression, it is done *immediately*.

4.4.4 Expect

The expect event asserts that a given signal has a given value at a given time. There are several forms of the expect primitive:

- Simple expect - @
- Full expect - @@
- Restricted - @@@

The general syntax for an expect statement is:

```
expect_operator delay window expect_list;
```

expect_operator - The *expect_operator* must be either @, which specifies a simple expect, @@, which specifies a full expect, or @@@, which specifies a restricted expect. Each form is discussed in detail in subsequent sections.

delay - The *delay* specifies the number of cycles that pass before the signal is driven. It is in the form @*n* where *n* is the number of cycles. A delay *must* be specified. For immediate checking, use a delay value of 0 cycles.

window - The *window* specifies how long the check is made. It must be in the form ,*n* where *n* is the number of cycles for which the check is made. The comma (,) immediately follows the delay. Any expect with a defined window is called a floating expect.

expect_list - The *expect_list* is any number of expressions (separated by commas) using the equal to (==) and not equal to (!=) operators. The bit value 'X' is treated as a "don't care" in expressions. The general form is *signal_name operator expression*. Signal names can be interface or port signals, and they can include subfields. If multiple expressions are declared, they are all sampled on the clock edge of the first signal in the list.

Note – The expect primitive is a blocking primitive. It blocks until the expect is satisfied or a simulation error is generated. There is not a non-blocking form of the expect primitive.

4.4.4.1 Simple Expect

The simple expect checks that a given signal has a specific value at a given time. The syntax is:

```
@ delay window expect_list;
```

If the signal value does not match the expression when the check is made, a simulation error is generated. If a subfield within the signal is specified, all other bits in the signal are ignored and only those specified are checked against the expression.

Multiple expressions can be defined in the *expect_list*. If multiple expressions are defined, the expect is satisfied if ALL of the conditions are satisfied at the time of the sample.

Expressions can also be separated by the *or* keyword. In that case, the expect is satisfied if any of the conditions is satisfied at the time of the check.

Note – You cannot mix comma separated expression lists with *or* lists.

If a window is specified, the expect is a floating expect. The check is made for the duration of the window. If the signal value matches the expression at any time within the window, the expect is satisfied. If the signal value does not match the expression within the window, a simulation error is generated. If multiple expressions are defined, the expect is satisfied when all the conditions are satisfied simultaneously. If the *or* keyword is used, the expect is satisfied as soon as any of the conditions is satisfied.

These are some examples of the expect statement:

```
@1 bus.data[7:4] == 4'b0101;
@0,10 bus.data == 4'bx0x1;
@1 bus.data == 4'b0010, bus.addr != 4'b0001;
@2,20 bus.data == 4'b0010, bus.addr != 4'b0001;
@2,20 bus.data == 4'b0010 or bus.addr == 4'b0001;
```

The first example expects the signal data to be equal to 0101 after 1 cycle.

The second example expects the signal data to be equal to 1011, 1001, 0011, or 0001 within 10 cycles.

The third example expects the signal data to be 0010 and the signal addr not to be 0001 after 1 cycle.

The fourth example expects the signal data to be 0010 and the signal addr not to be 0001 within 2 to 20 cycles.

The fifth example expects the signal data to be 0010 or the signal addr to be 0001 within 2 to 20 cycles.

4.4.4.2 Full Expect

Full expects check that a signal has a given value over the entire length of a given interval. The syntax is:

```
ee delay window expect_list;
```

Full expects behave in the same manner as simple expects with one exception. The signal value must match the expression over the entire course of the defined window. If the signal does not match during any part of the interval, the expect is not satisfied and a simulation error is generated.

This is an example of a full expect statement:

```
ee5,100 bus.data[7:4] != b'0101, bus.addr == 8'h88;
```

This example is satisfied if the signal data is not equal to 0101 and the signal addr is equal to 88 over the entire interval, 5 to 100 cycles.

4.4.4.3 Restricted Expect

Restricted expects check that a signal's value changes to a given value on the first signal change. The syntax is:

```
eee delay window expect_list;
```

Restricted expects check that the signal matches the expression on the first signal change within the window. If the signal value does not match the expression after the first signal change, a simulation error is generated. If the signal value matches the expression at the start of the window, the expect is satisfied immediately. If multiple expressions are defined using an AND construct, all of the conditions must be satisfied at the time of the first change to any of the signals, or else a simulation error is generated.

These are some examples of restricted expects:

```
eee0,100 bus.addr == 8'h01;
eee0,100 bus.data == 8'h88, bus.addr == 8'h01;
```

The first example is satisfied if the first change of signal addr is to 01.

The second example is satisfied if the first change of signal data is 88 and the signal addr changes to 01 simultaneously (or if it is already 01). The second example is also satisfied if the first change of signal addr is 01 and the signal data changes to 88 simultaneously (or if it is already 88).

4.4.4.4 Strong and Soft Expects

There are two strengths of expects: strong (default) and soft. The syntax to declare a soft expect is:

```
expect_operator delay window expect_list soft;
```

Soft expects do not generate simulation errors when they are not satisfied. Instead, they set an error flag and the simulation continues. You can check if an error flag has been set using the `flag()` system function (see Section 7.3.1, "Error Handling").

The `soft` keyword can be used with any of the expect primitives.

4.4.4.5 Void Expects

Expects are blocking in that if the Vera thread that executes them is not at the appropriate clock edge, it blocks until the appropriate clock edge occurs. If you want to wait until this clock edge without expecting the signal, use the `void` construct. The syntax is:

```
@delay signal_name == void;
```

For example:

```
@1 iport0.packet == void;
```

4.4.5 Value Change Alert (VCA)

The value change alert (VCA) checks that no unexpected event occurs on a signal during the simulation. VCAs are declared within the interface specification. The syntax to declare a VCA is:

```
signal_direction signal_width signal_name signal_type skew vca q_value;
```

q_value - The *q_value* is the quiescent state of the signal. The valid *q_values* are listed in Table 4-3.

Table 4-3 VCA *q_value*

<i>q_value</i>	Definition
r1	Signal returns to 1 (pullup)
r0	Signal returns to 0 (pulldown)
rz	Signal returns to z (tri)
rx	Signal returns to x (forced or driven)
nr	Signal holds last value (all transitions are flagged)
sr1	Signal returns to 1 (pullup)
sr0	Signal returns to 0 (pulldown)
srz	Signal returns to z (tri)
srx	Signal returns to x (forced or driven)
snr	Signal holds last value (all transitions are flagged)
gr1	Signal returns to 1 (pullup)
gr0	Signal returns to 0 (pulldown)
grz	Signal returns to z (tri)
grx	Signal returns to x (forced or driven)
gnr	Signal holds last value (all transitions are flagged)

The *q_value* for a vector applies to all bits in the vector. Thus an 8-bit bus defined as *r1* has the *q_value* 8'hff, *r0* would have a *q_value* 8'h00, *rz* 8'hzz, *rx* 8'hxx, and *nr* would have a *q_value* equal to whatever the previous value was.

The *q_values* *r1*, *r0*, *rz*, *rx*, and *nr* are asynchronous detects and detect any asynchronous changes except zero-delay glitches. The *q_values* *sr1*, *sr0*, *srz*, *srx*, and *snr* are synchronous detects and only detect errors at synchronous clock edges. The *q_values* *gr1*, *gr0*, *grz*, *grx*, and *gnr* are glitch detects and detect all asynchronous changes including zero delay glitches.

You only need to define *q_values* for those signals that have the VCA enabled for them.

Note – The VCA cannot be used with output signals.

This is an example of a signal declaration with the VCA specification:

```
input b PSAMPLE depth 1 vca rz;
```

Declaring a *q_value* in the interface specification does not enable the VCA for a signal. By default, all signals are VCA-disabled. To enable and disable the VCA for a signal or interface, use the system task *vca()*.

vca()

The *vca()* system task enables and disables VCAs. The syntax is:

```
vca(switch [, signal_name]);
```

switch - The *switch* must be either **ON**, which enables the VCA for the specified signal, or **OFF**, which disables the VCA for the specified signal. The default value is **ON**.

signal_name - The *signal_name* can either be the name of an interface, which enables or disables *all* VCAs within the interface, or it can be the name of an individual signal in the form *interface_name.signal_name*.

This is an example of the *vca()* system task:

```
vca(ON, myinterface);
vca(OFF, myinterface.signal1);
```

When a VCA is enabled for a particular signal, Vera checks that all changes to signals are to the quiescent state (*q_value*) unless there is a matching expect on the next sampling edge or the signal is explicitly driven to the new state.

After a change is made, the VCA does not drive the signal back to the quiescent state. To return the signal to its quiescent state, you must explicitly drive it.

Note that for pipelined signals, VCA reports are delayed. This is because Vera must first find out if there are any expects in the pipelined version of the signal that affect whether a value change is expected or not. So, Vera must wait until all of the cycles specified have been executed, and then it produces applicable VCAs.

Note – VCAs are implemented such that they do not slow down the simulation in any significant way.

4.4.6 Implicit Synchronization

The drive, sample, and expect primitives perform implicit synchronization to the interface CLOCK. That means that the clock is advanced only when it is necessary to perform the next signal operation. Take the following interface definition as an example:

```
interface foobus {
    output reset_l NHOLD;
    input strobe_l PSAMPLE;
    output ack_l NHOLD;
    inout data PSAMPLE NHOLD;
    input clock CLOCK;
}
```

In this interface, output signals are driven at the negative edge of the interface clock, and input signals are sampled at the positive edge of the interface clock. Thus, the following code advances the simulation cycle a half cycle per statement even though a delay is not specified.

```
reset_l = 1'b1;
strobe_l == 1'b1;
ack_l = 1'b0;
strobe_l == 1'b0;
```

The first signal is driven on the negative clock edge. The second signal is sampled on the positive clock edge. The third signal is driven on the negative clock edge. The fourth signal is sampled on the positive clock edge.

The description gets more complicated when delay values are used to generate proper timing with respect to different edges. To avoid this, use the same edge for inputs and outputs, with appropriate output skews. For example:

```
interface foobus {
    output reset_l PHOLD #2;
    input strobe_l PSAMPLE;
    output ack_l PHOLD #2;
    inout data PSAMPLE PHOLD #2;
    input clock CLOCK;
}
```

4.5 Asynchronous Operations

By default, drives, samples, and expects are relative to a clock edge (specified in the interface specification). However, the HDL side of the simulation may be using very detailed timing constructs. Vera provides the `async` and `delay` constructs to allow detailed timing down to the HDL timestep.

4.5.1 async

The **async** optional modifier specifies that the operation happen immediately, without waiting for the edge specified in the interface. It can be used with synchronization operators, drives, samples, and expects. The syntax is:

```
Sync: @ (signal_name async);

Drive: signal_name range drive_operator expression async;

Sample: variable = signal_name async;

Expect: @ expect_list async;
```

The synchronization construct allows you to act exactly on the specified edge rather than waiting for the corresponding sampling edge.

The drive, sample, and expect constructs force the operation immediately instead of waiting for the specified edge.

Note – Drive skews specified for the signal in the interface specification also apply to **async** drives. If you need to drive the signal precisely when the drive is issued, do not specify any skew for that signal in the interface specification.

If a signal has a VCA enabled for it, driving it with the **async** option results in a VCA error. Disable any active VCAs when driving signals asynchronously.

These are examples of **async** statements:

```
@(posedge main_bus.request async)
memsys.data[3:0] = 4'b1010 async;
data[2:0] = main_bus.data[2:0] async;
@main_bus.data[7:4] == 4'b0101 async;
```

4.5.2 Sub-Cycle Delays

Vera provides the **delay()** system task to block Vera while a specified amount of time elapses on the HDL side of the simulation. The syntax for the **delay()** system task is:

```
delay(time);
```

time - The *time* specifies the length of the delay. It is in the same timing units being used by the HDL.

This is an example of the **delay()** system task:

```
@(posedge CLOCK);
delay(5);
function1();
...
```

This example synchronizes to the positive edge of `CLOCK`. Then it advances the simulation time 5 time ticks. `Function1` executes 5ns after the clock edge.

Note – This does not work with Vera-CS because there is no HDL design generating a clock.

4.6 Subroutines

Vera supports two means of encapsulating often-executed program fragments: functions and tasks. All functions and tasks are re-entrant and can be called recursively.

Vera subroutines cannot be nested. This means that all subroutine declarations must be done at the top level. To ease the name space problem, Vera supports local subroutines and separate compilation. You can also declare external subroutines. These separately compiled Vera object files are linked at simulation time (see Section 18.6, “Modular Compilation”).

4.6.1 Functions

Functions are provided for implementing mathematical functions containing some number of arguments and one return value. Functions can be used in expressions in order to perform frequently used calculations, or to encapsulate the calculation. The syntax to declare a function is:

```
function data_type function_name (arguments) {statements;
```

data_type - The *data_type* can be any of the valid Vera data types. The value returned will be of the same data type that the function is declared with.

function_name - The *function_name* is the name by which the function is called throughout the program.

arguments - The *arguments* are the variables, including the data types, that are passed to the function when the function is called. All data types can be passed, including ports. Array arguments can be regular or associative, as well as `var`. Array arguments are strongly typed. Array type, width, and size must match exactly between the declaration and the call. Multiple arguments are separated by commas.

statements - The *statements* can be any Vera statement, including function calls, timing modifiers, and variable assignments.

Functions are designed to return a single value. They can return values of any data type as well as data structures. To set the return value, assign the name of the function a value somewhere within the function declaration.

This is an example function declaration:

```

function bit [3:0] even_byte_parity (bit [31:0] data)
{
    bit [3:0] tmp;

    tmp[3] = ^data[31:24];
    tmp[2] = ^data[23:16];
    tmp[1] = ^data[15: 8];
    tmp[0] = ^data[ 7: 0];
    even_byte_parity = tmp;
}

```

This example declares the function `even_byte_parity` with the argument `data`. Note the final line of the function, which contains the line that sets the return value.

Functions can be called in expressions from within the main program or from within other functions. The syntax to call a function is:

```
variable = function(arguments);
```

For example:

```
parity = even_byte_parity(Data);
```

By default, function names are global. Functions declared as local can only be used in the file where they are defined. To invoke a function defined in another file, you must use the `extern` declaration (see Section 4.6.7, "External Definitions"). The syntax to declare a local function is:

```
local function data_type function_name (arguments){statements;}
```

For example:

```
local function bit[3:0] g_decode ( integer i );
```

4.6.1.1 Discarding Function Return Values

Vera enforces the use of function return values. Calling a function as if it has no return value results in compilation errors. To explicitly discard a function's return value, use the `void` construct. The syntax is:

```
void = function(arguments);
```

4.6.2 Tasks

Tasks are identical to functions except they do not return a value. The syntax to declare a task is:

```
task task_name (arguments){statements;}
```

task_name - The *task_name* is the name by which the task is called throughout the program.

arguments - The *arguments* are the variables, including the data types, that are passed to the function when the function is called. All data types can be passed, including ports. Array arguments can be regular or associative, as well as `var`. Array arguments are strongly typed. Array type, width, and size must match exactly between the declaration and the call. Multiple arguments are separated by commas.

statements - The *statements* can be any Vera statement, including function calls, timing modifiers, and variable assignments.

This is an example task declaration:

```
task handshake_port0(bit direction, bit [7:0] data)
{
    @0,1000 port0.req == 1'b1;
    port0.ack = 1'b1;
    @1 port0.ack <= 1'b0;
    if(direction) port0.data = data;
    else port0.data == data;
}
```

Tasks can be invoked as statements. The syntax to invoke a task is:

```
task_name(arguments);
```

For example:

```
print_data(new_data);
```

By default, task names are global. Tasks declared as local can only be used in the file where they are defined. To invoke a task defined in another file, you must use the `extern` declaration (see Section 4.6.7, "External Definitions"). The syntax to declare a local task is:

```
local task task_name (arguments){statements;}
```

For example:

```
local task print_data (bit[7:0] data)
{
    printf("Local data = %h", data);
}
```

4.6.3 Return

Normally, functions and tasks return control to the caller after the last statement of the block is executed. Vera provides the `return` statement to manually pass control back to the caller. The syntax is:

```
return;
```

When the **return** statement is executed, the subprocess is terminated as if it had been exited normally. If the **return** statement is executed in a function before a value has been assigned, an undefined value is returned.

If a **return** statement is executed at the top code level, the simulation is terminated.

4.6.4 Breakpoint

Vera provides the **breakpoint** statement to stop the simulation and return control to the HDL. The syntax is:

```
breakpoint;
```

When a **breakpoint** statement is executed, Vera terminates the task or function immediately and returns control to the HDL. You must restart the simulation from the HDL command line. Simulation time continues while the process is terminated. So be careful when using breakpoints because the simulation time is not handled precisely on restart.

If a **breakpoint** statement is executed at the top code level, the simulation is terminated.

When a **breakpoint** statement is executed, the Vera debugger is launched.

4.6.5 Static Variables

By default, variables are local to the function or task that uses them. They are allocated when the function or task is called. This construct allows tasks and functions to be re-entrant and recursive.

If you want a variable to be shared across all invocations of a function or task, use the **static** declaration. The syntax to declare a static variable is:

```
static data_type variable_name;
```

Any data type can be declared as a static variable.

Note – In the case of concurrent accesses, there may be races if multiple threads assign to the same static variable.

4.6.6 Subroutine Arguments

Vera provides two means of accessing arguments in functions and tasks: “call by value” and “call by reference.”

4.6.6.1 Call by Value

"Call by value" is the default method that arguments are accessed in functions and tasks. Each subroutine retains a local copy of the argument. If the arguments are changed within the subroutine, the changes do not affect the caller.

4.6.6.2 Call by Reference

In a "call by reference," functions and tasks access the specified variables passed as arguments directly. The syntax to pass a subroutine argument by reference is:

```
subroutine (var data_type variable);
```

In a "call by reference," subroutines operate directly on the var arguments. The caller sees any changes to variables made within the subroutine. Variables of any type can be passed by reference.

This is an example of a "call by reference":

```
task IO_read_indirect ( bit[63:0] addr, var bit[31:0] io_data )
{
    // ... (modifies both addr and io_data)
    // ... (only the change in io_data will be seen by caller)
}

// caller
IO_read_indirect (my_addr, my_data );
```

In this example, the variable *io_data* is passed by reference. The task modifies all of the arguments passed, but only the change made to *io_data* is seen outside the task.

4.6.6.3 Default Arguments

To handle common cases or allow for unused arguments, Vera allows you to define default values for each scalar argument. The syntax to declare a default argument in a subroutine is:

```
subroutine(arg=default_value){statements}
```

default_value - The *default_value* can be any expression visible at the current code level. It can include any combination of constants, global variables, and functions.

When the subroutine is called, you can omit an argument that has a default defined for it. Use an asterisk (*) as a placeholder in the subroutine call. If an asterisk is used for a variable that does not have a default value, a compilation error occurs.

This is an example of a subroutine with default arguments:

```
task foo(integer i = 0, integer k, bit[5:0] data = 6'b0) {
    //..
}
```

```
foo(100, 5, *);
foo(*, 5, 6'b000111);
```

This example declares a task `foo` with default arguments. The first call to `foo` is equivalent to `foo(100, 5, 6'b0)`. The second call to `foo` is equivalent to `foo(0,5,6'b000111)`.

4.6.6.4 Optional Arguments

To allow subroutines to evolve over time without having to change all of the existing calls, Vera supports optional arguments. Optional arguments are encapsulated by parentheses, and they must have default values. The syntax is:

```
subroutine( (optional_arg1), ((optional_arg2)), (((optional_arg3))) )
{statements}
```

Any number of additional optional arguments can be created. The number of nested parentheses determines the depth level of the optional argument. For example:

```
((x=1)), ((y=1)) // Both arguments are level 2
({x=1, y=1}) // Both arguments are level 2
((x=1),((y=1))) // x is level 2, y is level 3
```

When a subroutine with optional arguments is called, the parameters fill up the lowest level arguments first. When the lowest level is satisfied, higher levels are then filled in ascending order. Note that the order of the arguments does not depend on the nesting level. For example:

```
task my_task(a, ((b=1)), (c=1));
```

This example declares a task with the required argument `a`, the level one argument `c`, and the level two argument `b`. These are examples of task calls using this declaration:

```
my_task(); // Illegal because the argument a has no default
my_task(1); // Calls task with a=1
my_task(1,2); // Calls task with a=1 and c=2
```

Note that the arguments are filled according to their nesting level. However, if enough arguments are passed to fill all of the optional arguments, the arguments are assigned values in order. For example:

```
my_task(1,2,3); // Calls task with a=1, b=2, and c=3
```

4.6.7 External Definitions

External definitions allow you to use multiple source files. This lets you compile large functions separately, which facilitates debugging.

4.6.7.1 Declaring External Subroutines

You can create subroutines in multiple source files. If you use multiple source files, you must declare the subroutines as external at the top level. The syntax to declare a subroutine as external is:

```
extern subroutine (arguments);
```

Alternatively, you can generate a header file when you compile the source code by using the **-h** switch (see Section 18.4, "Compile Options"). The generated header file contains all of the **extern** statements for that module. If you include this file, you do not need to manually declare the subroutines as **extern**.

Note – When using external subroutines, the argument types that are passed must match exactly. So take extra care when passing arguments to external subroutines.

4.6.7.2 External Default Arguments

The default values can be set locally, and independently, for each compilation unit using **extern** declarations with default values. This allows writing a general library, which can then be customized for a particular user or testbench by using include files with different defaults.

For example, the task `foo` may be defined in a separate library, which is compiled independently. The Vera file in which the task `foo` will be used must declare `foo` as being external, and in this **extern** declaration, you can set default values:

```
// file A (library)
task foo (integer i, k, bit[5:0] data) {
    // foo definition
}

// file B (testbench)
extern task foo ( integer i = 10, integer k, bit[5:0] data=6'b111111 );

task xyz () {
    foo (*, 5, *);
    // ...
}
```


Example 1: Vera Basics

This is an example of an arbiter. The example includes these features:

- File inclusion and text macros
- Interface specification
- Ports and binds
- Signal operation
- Subroutines

This example includes these files:

- *arb.if.vr*
- *signal_drive.vr*
- *RUN* (a run script)

arb.if.vrh

```
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE

interface arb
{
    output [1:0] bus_req PHOLD OUTPUT_SKEW;
    input [1:0] bus_gnt INPUT_EDGE vca r0; // vca monitor
    output [31:0] mem_addr PHOLD OUTPUT_SKEW;
    inout [15:0] mem_data INPUT_EDGE PRZ OUTPUT_SKEW vca rz; // vca monitor
    output oe PHOLD OUTPUT_SKEW;
    output ce PHOLD OUTPUT_SKEW;
    input ack INPUT_EDGE vca r0; // vca monitor
    input clk CLOCK;
}

port generic_port
{
    br;
    bg;
    addr;
    data;
    oe;
}
```

```

        ce;
        ack;
    }

    bind generic_port port_a
    {
        br arb.bus_req[0];
        bg arb.bus_gnt[0];
        addr arb.mem_addr;
        data arb.mem_data;
        oe arb.oe;
        ce arb.ce;
        ack arb.ack;
    }

    bind generic_port port_b
    {
        br arb.bus_req[1];
        bg arb.bus_gnt[1];
        addr arb.mem_addr;
        data arb.mem_data;
        oe arb.oe;
        ce arb.ce;
        ack arb.ack;
    }

```

signal_drive.vr

```

#include <vera_defines.vrh>
#include "arb.if.vrh"

program signal_test
{
    bit [15:0] write_data_a[4], write_data_b[4], read_data_a[4],
        read_data_b[4];

    // reset the request

    @0 port_a.$br <= 1'b0 ;
    @0 port_b.$br = 1'b0 ;

    // turn vca on for unexpected signal changes

    vca ( ON, arb );

    // initiate burst write with both port_b and port_a at the same time

```



```

fork
{
// port b
integer i;
for ( i = 0; i < 4; i++ )
    write_data_b[i] = random ();
do_write with port_b ( 32'h0000_0008, write_data_b );
}
{
// port a
integer i;
for ( i = 0; i < 4; i++ )
    write_data_a[i] = random ();
do_write with port_a ( 32'h0000_0004, write_data_a );
}
join all

// initiates burst read with both port_b and port_a at the same time
// and verify the results

fork
{ //concurrent block
// port_b
integer i;
do_read with port_b ( 32'h0000_0008, read_data_b );
for ( i = 0; i < 4; i++ )
{
    if ( read_data_b[i] != write_data_b[i] )
        error ( "port_b transactions gives inconsistent data!\n");
    else
        printf ("port_b data %0d OK!\n", i );
}
} // end concurrent block
{ //concurrent block
// port_a
integer i;
do_read with port_a ( 32'h0000_0008, read_data_a );
for ( i = 0; i < 4; i++ )
{
    if ( read_data_a[i] != write_data_a[i] )
        error ( "port_a transactions gives inconsistent data!\n");
    else
        printf ("port_a data %0d OK!\n", i );
}
} // end concurrent block

```

```

    join all
    printf ("\n----- All Finished ----- \n");
} //end program

// do_read : request bus ctrl from arbiter and do burst read at the
// addr

task do_read with generic_port ( bit [31:0] addr, var bit [15:0] data[4] )
{
    @1 $br = 1'b1; // start requesting for bus
    @1, 12 $bg == 1'b1; // bus should be grant within 12 cycles
    @0 $br = 1'b0; // stop the request immediately
    @2 $bg == 1'b0; // grant signal should be de-asserted
    @0 $addr <= addr; // drive the addr, oe, ce bits
    $oe = 1'b1;
    $ce = 1'b1;
    @2, 10 $ack == 1'b1; // ack should have come within 8 cycle
    fork
    { // concurrent block
        @0, 3 $ack == 1'b1; // ack should stay for 4 cycles
        @0 $oe <= 1'b0; // at the end, deassert oe and ce
        @0 $ce = 1'b0;
    } // end concurrent block
    { // concurrent block
        integer i;
        for (i = 0; i < 4; i++) // latch the data for 4 consecutive cycles
        {
            @(posedge CLOCK);
            data[i] = $data;
        } //end for
    } // end concurrent block
    join all
    @1 $oe = void; // void drive to kill one more cycle
} //end do_read

// do_write : request bus ctrl from arbiter and do burst write at
// the addr

task do_write with generic_port ( bit [31:0] addr, bit [15:0] data[4] )
{
    @1 $br = 1'b1; // start requesting for bus
    @1, 12 $bg == 1'b1; // bus should be grant within 12 cycles
    @0 $br = 1'b0; // stop the request immediately
    @2 $bg == 1'b0; // grant signal should be de-asserted
    @0 $addr <= addr; // drive the addr, oe, ce bits
    $oe = 1'b0;

```

```
$ce = 1'b1;
@2, 10 $ack == 1'b1; // ack should have come within 8 cycles
fork
{
    @0, 3 $ack == 1'b1; // ack should stay for 4 cycles
    @0 $ce = 1'b0;
}
{
    integer i;
    for (i = 0; i < 4; i++) // drive the data for 4 consecutive cycles
    {
        $data = data[i] async;
        @(posedge CLOCK);
    } //end for
}
join all
@1 $oe = void; // void drive to kill one more cycle
} //end do_write
```

RUN

```
#!/bin/csh -f
rm -f core *.vro *.vshell >& /dev/null
vera -cmp -g signal_drive.vr
```

104 Example 1: Vera Basics

5. Sequential Control

This chapter discusses the Vera constructs used for sequential flow control. It includes these sections:

- If-else Statements
- Case Statements
- Randcase Statements
- Repeat Loops
- For Loops
- While Loops
- Break and Continue

5.1 If-else Statements

Vera supports if-else statements as the general form of selection statements. The syntax to declare an if-else statement is:

```
if (condition) if_statement else else_statement;
```

condition - The *condition* can be any valid Vera expression.

statement - Either *statement* can be any valid Vera statement or block of statements. If a code block is used, the entire block is executed.

If the *condition* evaluates to true, the *if_statement* is executed. If it is evaluated to false, the *else_statement* is executed.

The else statement can be omitted. If the else statement is omitted, the conditional is evaluated and the *if_statement* executes only if it evaluates to true. Otherwise, the program continues execution with the first line after the if statement.

Vera supports nested if-else statements.

This is an example of an if-else statement:

```
if (operator==0) y=a+b;
else if (operator==1) y=a-b;
else if (operator==2) y=a*b;
else y='bx';
```

This example uses several if-else statements. Note that the final else statement is associated with the statement immediately preceding it.

Vera also supports ? operator to replace if-else statements. The syntax is:

```
condition ? if_statement : else_statement;
```

This construct behaves exactly as the if-else construct. The *condition* is evaluated. If it evaluates to true, the *if_statement* is executed. If it does not evaluate to true, the *else_statement* is executed.

5.2 Case Statements

Vera provides the **case** statement for multi-way branching. The syntax to declare a **case** statement is:

```
case (primary_expression) {
    case1_expression : statement;
    case2_expression : statement;
    ...
    caseN_expression : statement;
    default : statement;
}
```

expression - The *expressions* can be any valid Vera expression.

statement - The *statement* can be any valid Vera statement or block of statements. If a code block is used, the entire block is executed.

The *primary_expression* is evaluated. The value of the *primary_expression* is successively checked against each *case_expression*. When an exact match is found, the *statement* corresponding to the matching case is executed, and control is then passed to the first line of code after the case block. If other matches exist, they are not executed. If no match is found, the **default statement** is executed.

All case expressions must be the same bit length. 'X' and 'Z' values are actual signal values and are not ignored.

This is an example case block:

```
case ( bus[3:0] ) {
    4'b00ZZ: packet = NULL;
    4'b0001: packet = READ;
    4'b0010: packet = WRITE;
    4,b00XX: packet = UNKNOWN;
    default:
    {
        printf( "Error: illegal packet %h detected\n", bus[3:0] );
        packet_error();
    }
}
```

If you want to use 'X' or 'Z' as a "don't care", use the **casex** or **casez** statements. 'X' values are ignored when using **casex**, and 'Z' values are ignored when using **casez**.

5.3 Randcase Statements

The **randcase** statement creates a block of statements, one of which is executed randomly. The syntax to declare a **randcase** block is:

```
randcase {  
    weight1 : statement1;  
    weight2: statement2;  
    ...  
    weightN : statementN;  
}
```

weight - The branch *weight* can be any valid Vera expression, including a constant. The expression is evaluated every time a **randcase** is executed.

statement - The *statement* can be any valid Vera statement or block of statements. If a code block is used, the entire block is executed.

When the **randcase** statement is executed, a statement is randomly selected from the block. different weights can be used to change the probability that any given statement is selected. The probability that any single statement is selected is determined by *weight/total_weight*.

Randcase statements can be nested.

This is an example of a **randcase** block:

```
randcase {  
    10: i=1;  
    20: i=2;  
    50: i=3;  
}
```

This example defines a **randcase** block with the specified weights. There is a .125 probability that the first statement is executed, a .25 probability that the second statement is executed, and a .625 probability that the third statement is executed.

5.4 Repeat Loops

The **repeat** loop executes a statement a fixed number of times. The syntax to declare a **repeat** loop is:

```
repeat (expression) statement;
```

expression - The *expression* can be any valid Vera expression, including constants.

statement - The *statement* can be any valid Vera statement or block of statements. If a code block is used, the entire block is executed.

Repeat statements can be used to repeat any statement a fixed number of times. The value of the *expression* is evaluated before the repetitions start. Changing a variable within the *expression* does not change the number of loops to be executed.

Repeat statements are often used to implement a wait or pause in the simulation. For example:

```
repeat (10) @posedge CLOCK;
```

This example pauses the simulation 10 clock cycles.

5.5 For Loops

Vera provides the **for** construct for creating loops. The syntax to declare a for loop is:

```
for (initial; condition; increment) statement;
```

initial - The *initial* is an assignment statement used to set the loop control variables.

condition - The *condition* can be any valid Vera expression.

increment - The *increment* defines how the loop control variable changes each time the loop is repeated. It can be any valid expression.

statement - The *statement* can be any valid Vera statement or block of statements. If a code block is used, the entire block is executed.

The for loop sets the initial value of the loop control variable. It evaluates the *condition*. If the condition is true, the loop executes a single time. When the loop finishes one iteration, the *update* expression is executed. Typically this expression changes the value of the loop control variable. Then the *condition* is checked again and the process continues. The loop continues as long as the *condition* evaluates to true. When it does not evaluate to true, the loop stops and control is passed to the first line of Vera code after the loop.

You can specify multiple variables in the *initial* statement, separating them with commas. Multiple variables can also be used in the *condition* expression. These variables (with their initialized values) are passed to the loop and can be used within the loop for loop control or in Vera expressions.

Vera does not allow assignments within the conditional. The conditional `c=1` is invalid. Instead, you must use `c==1`.

These are some examples of Vera for loops:

```
for(count=0; count<3; count=count+1)
    value=value+((a[count]) * (count+1));

for(count=0, done=0, i=0; i*count<125; i++)
    printf("Value i = %d\n", i);
```


5.6 While Loops

Vera provides the **while** construct for creating loops. The syntax to declare a while loop is:

```
while (condition) statement;
```

condition - The *condition* can be any valid Vera expression.

statement - The *statement* can be any valid Vera statement or block of statements. If a code block is used, the entire block is executed.

The loop iterates while the condition is true. When the condition is false, control passes to the first line of Vera code after the loop. The condition is checked at the top of each loop.

Vera does not allow assignments within the conditional. The conditional `c=1` is invalid. Instead, you must use `c==1`.

This is an example of a while loop:

```
operator = 0;
while (operator<5){
    operator=operator+1;
    printf("Operator is %d", operator);
}
```

This loop continues until *operator* equals 5. Each time through the loop, *operator* is increased by 1. The check is made at the top of each loop. After 5 passes through the loop, the loop ends, and control is passed to the first line of code after the loop.

If the *condition* is a non-zero constant, the loop becomes infinite. Infinite loops can only be broken using the **break** statement (see Section 5.7.1, "Break").

5.7 Break and Continue

Vera provides the **break** and **continue** statements for flow control within loops.

5.7.1 Break

The **break** statement is used to force the immediate termination of a loop, bypassing the normal loop test. The syntax to declare a break is:

```
break;
```

When the **break** statement is executed from inside a loop, the loop is immediately terminated and control passes to the first line of Vera code after the loop. If the **break** statement is executed outside of a loop, a syntax error is generated.

This is an example of the **break** statement:

```
while (test_flag) {  
    if (done) break;  
    ...  
}
```

This example breaks if the condition is satisfied. Control returns to the first line after the loop.

5.7.2 Continue

The **continue** statement forces the next iteration of a loop to take place, skipping any code in between. The syntax to declare a **continue** statement is:

```
continue;
```

In a **repeat** loop, the **continue** statement passes control back to the top of the loop. If the loop is complete, control is then passed to the first line of code after the loop.

In a **for** loop, the **continue** statement causes the conditional test and increment portions of the loop to execute.

In a **while** loop, the **continue** statement passes control to the conditional test.

This is an example of a **continue** statement:

```
for (i=0;i<10;i++) {  
    if (skip_loop) continue;  
    ...  
}
```

6. Concurrent Control

This chapter discusses how Vera handles concurrency. It explains how to model parallel, independent activities and details the Vera constructs used to control those concurrent threads. It includes these sections:

- Fork and Join
- Events
- Semaphores
- Regions
- Mailboxes
- Timeout Limit
- Backward Compatibility

6.1 Fork and Join

Fork/join blocks provide the primary mechanism for creating concurrent processes. The syntax to declare a fork/join block is:

```
fork
    {statement1;}
    {statement2;}
    {...}
    {statementN;}
join wait_option
```

statementN - The *statements* can be any valid Vera statement.

wait_option - The *wait_option* specifies when the code after the fork/join block executes. The *wait_option* must be **all**, **any**, or **none**. The **all** option is the default. Code after the block executes after all of the concurrent processes have completed. When the **any** option is used, code after the block executes after any single concurrent process is completed. When the **none** option is used, code after the block executes immediately, without waiting for any of the processes to complete.

The flow for a fork/join block is shown in Figure 6-1.

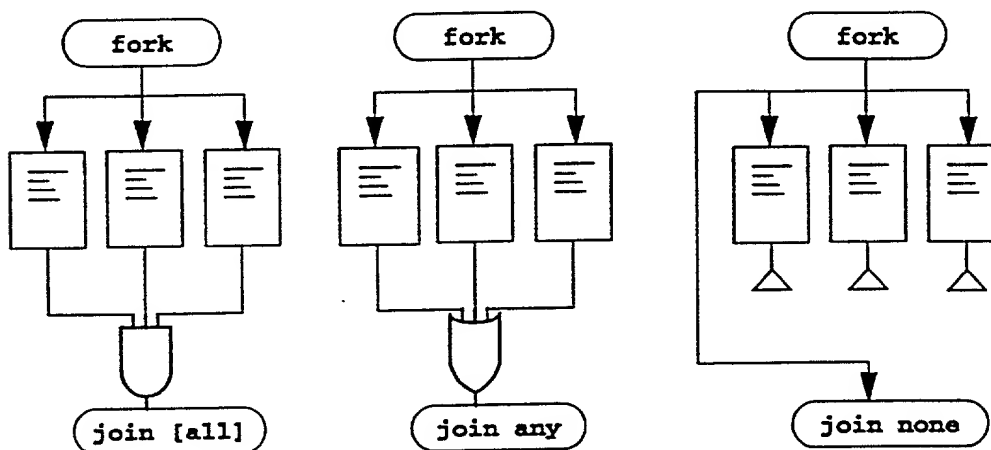


Figure 6-1 Fork/Join Flow Diagram

Note – When defining a fork/join block, do not encapsulate the entire fork inside braces ({}). Doing so results in the entire block being treated as a single process, and the code executes consecutively. For example, avoid this construct:

```
fork {
    {statement1;}
    {statement2;}
}
join
```

This is an example of a basic fork/join construct:

```
fork
{
    @1,100 foo_bus.ack == 1'b0;
    printf("First Block: foo_bus.ack is driven\n");
}

{
    @5 foo_bus.req = 1'b0;
    @1 foo_bus.req <= 1'b1;
    printf("Second Block: foo_bus.req is driven\n");
}

join
```

The concurrent block executes all the statements in parallel. The beginning of each statement is executed at the same point in time. Subsequent statements are executed based on any timing considerations within the process.

6.1.1 Fork and Join Control

Vera provides several constructs to control fork/join blocks. Vera uses the `wait_child()`, `wait_var()`, and `terminate` constructs to wait for the completion of processes and stop the execution of processes. Vera also includes the `suspend_thread()` system task to temporarily suspend threads.

6.1.1.1 `wait_child()`

The `wait_child()` system task is used to ensure that all child processes are executed before the Vera program terminates. The syntax is:

```
wait_child();
```

By default, Vera terminates a simulation when the end of the program is reached, regardless of the status of any child processes. Using the `wait_child()` task causes Vera to wait until all the child processes in the current context are completed before executing the next line of code.

This is an example of a program using the `wait_child()` construct:

```
program test
{
    start_monitors(); //Starts monitors that loop forever in background
    do_test(); //Performs the actual test
}

task start_monitors()
{
    fork
    {...}
    join none
}

task do_test()
{
    //Code to do testing.
    fork
    {...}
    join none //Creates child processes that take an indeterminate amount
              //of time to complete
    wait_child();
}
```

This example calls two separate tasks. The `do_test` task forks off several child processes that take an indeterminate amount of time to complete. The `wait_child()` call waits for the threads called in the `do_test` task to complete before executing subsequent Vera code. Note that the `wait_child()` call does not wait for any child processes created outside of its context.

6.1.1.2 wait_var()

The `wait_var()` system task blocks the calling process until one of the variables in its arguments list changes values. The syntax is:

```
wait_var(variables);
```

variables - The *variables* are one or more variables (separated by commas) of type integer, bit, string, array, or enumerated type.

The `wait_var()` task blocks the current process until one of the specified variables changes value. Only true value changes unblock the process. Reassigning the same value does not unblock. If more than one variable is specified, a change to any of the variables unblocks the process.

This is an example of the `wait_var()` task:

```
bit[7:0] data [100];
integer i;

fork
{
    wait_var(data[2]);
    printf("Data[2] has changed to: %d\n", data[2]);
}
{
    for (i=0;i<100;i++)
    {
        data[i]=random();
        @(posedge CLOCK);
    }
}
join
```

This example forks off concurrent processes. The first thread is suspended until the second element of array *data* is changed. The second process randomly changes the values within array *data*. When *data*[2] is changed, the first process prints its message.

6.1.1.3 terminate

The `terminate` statement terminates all active descendants of the process in which it was called. The syntax is:

```
terminate;
```

If any of the child processes have other descendants, the **terminate** command terminates them as well. If used at the top level, **terminate** terminates all child processes. When the main program is completed, Vera executes an implicit **terminate** statement.

This is an example of how **terminate** is used within a simple fork/join block:

```
task do_test()
{
    // Code to do testing
    fork
    {...}
    join any// Creates child processes that take an indeterminate amount of
           //time to complete
    // Code to do more testing
    terminate;
}
```

This example forks off several child processes within a task. After any of the child processes is complete, Vera continues to execute code. Before the task is completed, all remaining child processes are terminated.

6.1.1.4 suspend_thread()

The **suspend_thread()** system task is used to temporarily suspend the current thread. The syntax is:

```
suspend_thread();
```

The **suspend_thread()** system task temporarily suspends the current thread and allows other ready concurrent threads to run. When the other threads are done, the suspended thread resumes execution. For example:

```
for (i=0;i<10;i++)
{
    fork
    my_task(i);
    join none
    suspend_thread();
}
```

This example forks multiple threads calling **my_task**. The thread is forked, the task is called, and then the thread is suspended. The next iteration of the loop occurs and forks the next thread. That thread begins execution and is suspended. All 10 threads are created, begin execution, and are suspended. Control returns to the first thread, and then each is executed in sequence. Using this construct, you do not need to declare *i* as a **shadow** variable.

Note – Suspended threads execute after all other current threads execute. However, relative to simulation time, the thread is still executed concurrently with the other threads.

6.1.2 Shadow Variables

By default, all child processes have access to the parent's variables. However, if multiple processes independently write to the same variable, races can occur. To avoid races within fork/join blocks, Vera uses shadow variables. The syntax to declare a shadow variable is:

```
shadow data_type variable_name;
```

Using the **shadow** keyword forces the Vera compiler to create a copy of the variable local to each child process, which eliminates race conditions.

This is an example of how shadow variables are used:

```
shadow bit[31:0] address;
shadow bit[31:0] data;
integer i;

for (i=0;i<num_try;i++)
{
    repeat(random()%MAX_DELAY) @(CLOCK);
    address=random();
    data=random();
    printf("Random access with Addr:%h Data%h\n", address, data);
    fork
    {
        Memory_Access(address, data);
    }
    join none
}
```

This example declares the variables *address* and *data* as shadow variables. The for loop then randomizes these variables and forks off a child process that calls the function `Memory_Access` with *address* and *data* as parameters. The loop continues without waiting for the child process to complete because of the `join none` statement. The shadow variables must be used in this instance to ensure that the call to `Memory_Access` is made with variables that have not been re-randomized by subsequent loops.

6.2 Events

Events are variables that synchronize concurrent processes. When a sync is called, a process blocks until another process sends a trigger to unblock it. Events act as the go-between for triggers and syncs.

6.2.1 Triggers

Triggers are used to send events. The syntax to call a trigger is:

```
trigger([trigger_type,] event_name);
```

trigger_type - Table 6-1 lists the trigger types with a brief definition.

Table 6-1 Trigger Types

Trigger Type	Action
ONE_SHOT	Triggers pending syncs (default)
ONE_BLAST	Like ONE_SHOT, but it includes all syncs in the current cycle
HAND_SHAKE	Triggers the oldest pending sync, or queues request
ON	Turns on the event
OFF	Turns off the event

event_name - The *event_name* is the name of the event being triggered.

ONE_SHOT

The ONE_SHOT trigger is the default trigger type. If you use a ONE_SHOT trigger, any process waiting for a trigger receives it. If there is no process waiting for the trigger, the trigger is discarded.

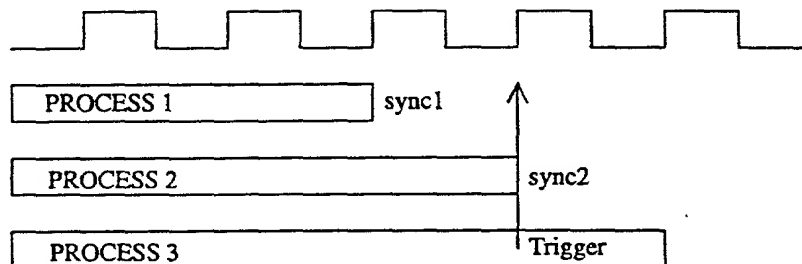


Figure 6-2 ONE_SHOT Triggers

There is a danger when using ONE_SHOT triggers. Figure 6-2 shows a trigger called in process 3. The trigger unblocks sync1. However, because the trigger and sync2 execute simultaneously, whether or not process 2 is unblocked depends on the execution order. The sync must be called before the trigger is executed when using ONE_SHOT triggers. If the sync is called after the trigger is executed, the process will wait indefinitely.

ONE_BLAST

ONE_BLAST triggers work just as **ONE_SHOT** triggers with the exception that they trigger any sync called within the current cycle, regardless of whether or not it was called before the trigger was executed. If a **ONE_BLAST** trigger is used for the situation diagrammed in Figure 6-2, all the processes are unblocked regardless of execution order.

HAND_SHAKE

HAND_SHAKE triggers unblock only one sync, even if multiple syncs are waiting for triggers. If a sync has already been called and is waiting for a trigger, the **HAND_SHAKE** trigger unblocks the sync. If no sync has been called when the trigger occurs, the **HAND_SHAKE** trigger is stored. When a sync is called, the sync is immediately unblocked and the trigger is removed.

ON

The **ON** trigger is used to turn on an event. An event cannot be used to synchronize concurrent processes if it is not turned on.

OFF

The **OFF** trigger is used to turn off an event. An event cannot be used to synchronize concurrent processes if it is turned off.

6.2.2 Sync

The `sync()` system task synchronizes statement execution to one or more triggers. The syntax to call the `sync()` task is:

```
sync(sync_type, event1, event2, ... eventN);
```

sync_type - The possible sync types and their definitions are listed in Table 6-2.

Table 6-2 Sync Types

Sync Type	Definition
ALL	Waits until all events are triggered
ANY	Waits until any event is triggered
ORDER	Waits until all events are triggered in the specified order
CHECK	Checks if all events are ON

eventN - The *event* is the event variable name on which the sync is activated.

ALL

The **ALL** sync type suspends the process until all of the specified events are triggered. For example:

```
sync(ALL, event_a, event_b, event_c);
```

This example suspends the thread until each of the events are triggered. Once they are triggered, the statement immediately following the `sync()` call is executed.

ANY

The **ANY** sync type suspends the process until any of the specified events is triggered. For example:

```
sync(ANY, event_a, event_b, event_c);
```

This example suspends the thread until any of the specified events is triggered. Once one of the events is triggered, the statement immediately following the `sync()` call is executed.

ORDER

The **ORDER** sync type suspends the process until all of the specified events are triggered in the given order. For example:

```
sync(ORDER, event_a, event_b, event_c);
```

This example suspends the thread until all of the specified events are triggered. As soon as an event is received out of order, the process unblocks and a simulation error occurs. Also, only the first event can be in the ON state when the `sync` is called. If both `event_a` and `event_b` are ON when the call is made, a simulation error occurs.

Note – Events that are set to NULL are always treated as if they were received in the correct order.

CHECK

The **CHECK** sync type is called as a function. It does not suspend the thread. It returns a 1 if the trigger is ON and a 0 if it is not. This sync type can only be used with ON and OFF trigger types. This sync type can only be used with a single event per call. For example:

```
if (sync(CHECK, event_a) )  
    printf("The event is ON.\n");
```

This `sync()` call returns a 1 if the event is on, and then prints the message. If the event is OFF, a 0 is returned.

6.2.3 Event Variables

Event variables serve as the link between triggers and syncs. They are a unique data type with several important properties.

6.2.3.1 Bidirectional Event Variables

Event variables are bidirectional variables when used as arguments in syncs and triggers. The same event variable can be used to pass and receive triggers. For example:

```
task foo (event trigger_a)
{
    printf("\nFOO syncing at cycle=%0d",get_cycle());
    sync(ALL, trigger_a); // Blocked: proceed after receiving trigger
    printf("\nFOO event trigger_a received at cycle=%0d",get_cycle());
    repeat (5) @(posedge CLOCK);
    printf("\nFOO triggering trigger_a at cycle=%0d", get_cycle() );
    trigger (trigger_a);
}

program trigger_play
{
    event trigger1;
    breakpoint;

    // top block code starts here

    fork
        foo(trigger1); // start foo and go on
    join none
    repeat(8) @(posedge CLOCK); //foo is blocked waiting for event trigger
    fork
        printf("\nPROGRAM triggering trigger1 @cycle=%0d", get_cycle());
        printf("\nPROGRAM This unblocks foo");
        trigger(trigger1); // unblock the waiting foo
    {
        repeat (7) @(posedge CLOCK);
        printf("\nPROGRAM syncing @cycle=%0d\n\n", get_cycle());
        sync (ALL, trigger1) ;// wait for foo to unblock me
    }
    join
    wait_child();
    printf("Trigger play done!");
}
```

This example declares the task `foo`, which is called in the main program. Then `foo` is called in a thread forked off from the main program. The program continues without waiting for the child process to complete. Because `foo` contains a `sync` within its definition, the child process blocks, waiting for a trigger. Then another fork is used to fork off a trigger, which unblocks the suspended `foo`. A second process in that fork then calls a `sync`. This `sync` occurs as `foo` is unblocked. `foo` then continues its execution, which includes the execution of the trigger that unblocks the final child process.

6.2.3.2 Disabling Events

If an event variable is assigned a null value, the event is ignored in subsequent `sync()` calls that may be waiting for a trigger on the event variable. For example:

```
event boo = null;
sync (ALL, boo);
```

The `sync` is immediately satisfied because of the null value assigned to `boo`.

6.2.3.3 Merging Events

When an event is assigned to another event, the two are merged, which causes triggers on either one to affect both. For example:

```
event boo, foo, woo;
boo = foo;
trigger(boo);
trigger(boo); // this will trigger foo, as well
trigger(foo); // this will trigger boo, as well
boo = woo;
foo = boo;
trigger(boo); // this will trigger foo and woo, as well
trigger(foo); // this will trigger boo and woo, as well
trigger(woo); // this will trigger boo and foo, as well
```

However, use caution when merging events. The assignment affects subsequent triggers and syncs. For example, if a process is blocked waiting for `event1` when you assign another event to `event1`, the `sync` will never unblock. For example:

```
fork
{
    while (1) {sync (ALL, foo);}
}

{
    while (1) {sync (ALL, boo);}
}

{
    foo = boo;
    while (1) {trigger (foo);}
}

join
```

This example forks off three concurrent threads. Each starts at the same time in the simulation. So, at the same time that threads 1 and 2 are blocked, thread 3 assigns the event `boo` to `foo`. This means that thread 1 can never unblock, because the event `foo` is now `boo`. To unblock both threads 1 and 2, the merger of `foo` and `boo` must take place before the fork.

6.3 Semaphores

A semaphore is a primitive operation used for mutual exclusion and synchronization.

The semaphore system functions are:

```
alloc(SEMAPHORE, semaphore_id, semaphore_count [, key_count]);  
semaphore_get(wait_option, semaphore_id, key_weight);  
semaphore_put(semaphore_id, key_weight);
```

6.3.1 Conceptual Overview

Conceptually, semaphores can be viewed as a bucket. When you allocate a semaphore, you create a virtual bucket. Inside the bucket are a number of keys. No process can be executed without first having a key. So, if a specific process requires a key, only a finite number of occurrences of that process can be in progress simultaneously. All others must wait until a key is returned to the virtual bucket.

6.3.2 `alloc()`

To allocate a semaphore, you must use the `alloc()` system function. The syntax is:

```
alloc(SEMAPHORE, semaphore_id, semaphore_count [, key_count]);
```

semaphore_id - The *semaphore_id* is the ID number of the particular semaphore being created. It must be an integer value. You should generally use 0. When you use 0, Vera automatically generates a semaphore ID.

semaphore_count - The *semaphore_count* specifies how many semaphores you want to create. It must be an integer value.

key_count - The *key_count* specifies the number of keys allocated to each semaphore.

The `alloc()` function returns the base semaphore ID if the semaphores are successfully created. Otherwise, it returns 0.

The maximum number of semaphores that can be created is determined by a runtime parameter. See Section 18.5, "Running Vera with HDLs."

6.3.3 `semaphore_get()`

To check that there are enough keys left in the semaphore, you must use the `semaphore_get()` system function. The syntax is:

```
semaphore_get(wait_option, semaphore_id, key_weight);
```

wait_option - The wait option can either be NO_WAIT or WAIT. The NO_WAIT option continues code execution if there are not enough keys available. The WAIT option suspends the process until there are enough keys available, at which time execution continues.

semaphore_id - The *semaphore_id* specifies which semaphore to get keys from.

key_weight - The *key_weight* specifies the number of keys being taken from the semaphore.

When the `semaphore_get()` function is called, it checks the specified semaphore for the number of required keys. If there are enough keys available, a 1 is returned and execution continues. If there are not enough keys available, a 0 is returned and the process is suspended depending on the wait option.

The semaphore waiting queue is FIFO based. By default, a process will wait at a semaphore without timing out. Users can set a time limit with the `timeout()` system task. See Section 6.5.5, "Mailbox Example."

If multiple semaphores are allocated, you can access the Nth semaphore using this method:

```
semID=alloc(SEMAPHORE, 0, 4, 2);
if (semaphore_get(WAIT, semID+2, 1))
    printf("The semaphore was successful.");
```

This example allocates four semaphores with IDs 0 to 3, each with two keys. Then it checks to see if there is a key in the third semaphore. If there is, a message is printed.

6.3.4 semaphore_put()

To put keys back into a semaphore, you must use the `semaphore_put()` system task. The syntax is:

```
semaphore_put(semaphore_id, key_weight);
```

semaphore_id - The *semaphore_id* specifies which semaphore to return the keys to.

key_weight - The *key_weight* specifies the number of keys being returned to the semaphore.

When the `semaphore_put()` system task is called, the specified number of keys is returned to the semaphore. If a process has been suspended to wait for a key, that process executes when enough keys have been returned.

6.3.5 Semaphore Example

This is an example of basic semaphore usage:

```
class gen
{
    local reg[2:0] bit_field_a;
    static integer gu = alloc(SEMAPHORE, 0, 1, 1);
    task m1()
```

```

    {
        printf("The value of guard %0d, in gen\n", gu);
    }
    task new()
    {
        bit_field_a = random();
    }
}

program test
{
    integer i, j;
    gen g1, g2, g3;

    printf("This is program test beginning.\n");
    g1=new;g2=new;g3=new;
    printf("The gnd in main %0d %0d %0d\n", g1.gu, g2.gu, g3.gu);
    fork
    {
        @(posedge CLOCK);
        semaphore_get(WAIT, g1.gu, 1);
        printf("This is g1 semaphore_get at cycle %0d\n", get_cycle());
        repeat (2) @(posedge CLOCK);
        semaphore_put(g1.gu, 1);
    }
    {
        @(posedge CLOCK);
        semaphore_get(WAIT, g2.gu, 1);
        printf("This is g2 semaphore_get at cycle %0d\n", get_cycle());
        repeat (2) @(posedge CLOCK);
        semaphore_put(g2.gu, 1);
    }
    {
        @(posedge CLOCK);
        semaphore_get(WAIT, g3.gu, 1);
        printf("This is g3 semaphore_get at cycle %0d\n", get_cycle());
        repeat (2) @(posedge CLOCK);
        semaphore_put(g3.gu, 1);
    }
    join all
}

```

This example creates class `gen` inside which a single semaphore is allocated with 1 key. Within the main program, `g1`, `g2`, and `g3` are instantiated. These are instances of the class `gen`. However, there is only one semaphore because of the static declaration in the semaphore allocation. Then the main program forks off three separate threads. Each thread tries to get a key from the semaphore. If it is unable,

it waits until a key is available. Once it gets the key, it prints a message, advances the simulation clock, and returns the key to the semaphore. In this way, each thread is suspended until a semaphore key is available, which can be used to prevent conflicts between threads.

6.4 Regions

A region is a mutual exclusion mechanism that guarantees that the requested values are unique in the simulation. This feature is provided mostly for random type simulations that may depend on the uniqueness of specific values such as addresses or data-IDs.

The region system functions are:

```
alloc(REGION, region_id, region_count);
region_enter(wait_option, region_id, value1, value2, ..., valueN);
region_exit(region_id, value1, value2, ..., valueN);
```

6.4.1 Conceptual Overview

Conceptually, regions can be viewed as a set of letters. First you allocate which letters are included in the set. These letters are the only letters from which words can be made. If one person uses the letters to spell CAT, no one else can spell TIN because the T is already in use. Once the T is returned, TIN can be created. Effectively, this ensures that data sets are unique, and it eliminates concurrent crossover.

6.4.2 alloc()

To allocate a region, you must use the **alloc()** system function. The syntax is:

```
alloc(REGION, region_id, region_count);
```

region_id - The *region_id* is the ID number of the particular region being created. It must be an integer value. You should generally use 0. When you use 0, Vera automatically generates a region ID.

region_count - The *region_count* specifies how many regions you want to create. It must be an integer value.

The **alloc()** function returns the base region ID if the regions are successfully created. Otherwise, it returns 0.

The maximum number of regions that can be created is determined by a runtime parameter. See Section 18.5, "Running Vera with HDLs."

6.4.3 region_enter()

The **region_enter()** system function checks to see if a particular region is in use. The syntax is:

```
region_enter(wait_option, region_id, value1, value2, ..., valueN);
```

wait_option - The wait option can either be **NO_WAIT** or **WAIT**. The **NO_WAIT** option continues code execution if the specified region is in use. The **WAIT** option suspends the process until the specified region is no longer in use.

region_id - The *region_id* specifies which region that is being entered.

valueN - The values are integer or bit vectors up to 64 bits, without X's or Z's. These values specify the unique region values.

The **region_enter()** system function checks the specified values against all region values for the specified region. If another process has entered the region with one or more of the values, then those values are in use, and the current region cannot use them. If none of the values are in use elsewhere, the function returns a 1, flags the values as in use, and passes control to the next line of code. If one or more of the values is in use elsewhere, the function suspends the current thread until the values become available, depending on the wait option.

The region waiting queue is FIFO based. By default, a process will wait at a region without timing out. Users can set a time limit with the **timeout()** system task. See Section 6.5.5, "Mailbox Example."

6.4.4 region_exit()

The **region_exit()** system task removes the specified values from the in-use state. The syntax is:

```
region_exit(region_id, value1, value2, ..., valueN);
```

region_id - The *region_id* specifies which region that is being exited.

valueN - The values are integer or bit vectors up to 64 bits, without X's or Z's. These values specify the unique region values.

When the **region_exit()** system task is called, the specified values are no longer in use and can be used in other regions. Any processes that are suspended waiting for region values execute when the region values are made available.

6.4.5 Region Example

This is an example of region usage:

```
task CPU(integer id, integer grant, integer regID)
{
    integer data, address;
    bit[31:0] randVar;

    repeat(256) {
        randVar=random();
        address=randVar[13:6];
        data=randVar[29:22];
        region_enter(WAIT, regID, address);
```

```

        @1 memsys.request[id]=1'b1;
        @2,20 memsys.grant==grant;
        writeOp(address, data);
        @1 memsys.request[id]=1'b0;
        @2,20 memsys.grant==2'b00;
        @1 memsys.request[id]=1'b1;
        @2,20 memsys.grant==grant;
        readOp(address,data);
        @1 memsys.request[id]=1'b0;
        @2,20 memsys.grant==2'b00;
        region_exit(regID, address);
        repeat(randVar[20:17])@(posedge memsys.clk);
    }
}

regID=alloc(REGION, 0, 1);
fork
    CPU(0, 2'b01, regID);
    CPU(1, 2'b10, regID);
join

```

This example repeats a loop 256 times. Within the loop, first a random 32 bit value is generated and assigned to *randVar*. Then a random address and random data are assigned to *address* and *data*. Next, the *region_enter()* call checks to see if the address is in use. If it is in use, the process suspends until the value is freed. When the value is not in use, execution continues until the region is exited.

6.5 Mailboxes

A mailbox is a mechanism to exchange messages between processes. Data can be sent to a mailbox by one process and retrieved by another.

The mailbox system functions are:

```

alloc(MAILBOX, mailbox_id, mailbox_count);
mailbox_put(mailbox_id, data);
mailbox_get(wait_option, mailbox_id [, dest_var [, check_option] ]);

```

6.5.1 Conceptual Overview

Conceptually, mailboxes behave like real mailboxes. When a letter is delivered and put into the mailbox, you can retrieve the letter (and any data stored within). However, if the letter has not been delivered when you check the mailbox, you must choose whether to wait for the letter or retrieve the letter on subsequent trips to the mailbox. Similarly, Vera's mailboxes allow you to transfer and retrieve data in a very controlled manner.

6.5.2 alloc()

To allocate a mailbox, you must use the `alloc()` system function. The syntax is:

```
alloc(MAILBOX, mailbox_id, mailbox_count);
```

mailbox_id - The *mailbox_id* is the ID number of the particular mailbox being created. It must be an integer value. You should generally use 0. When you use 0, Vera automatically generates a mailbox ID.

mailbox_count - The *mailbox_count* specifies how many mailboxes you want to create. It must be an integer value.

The `alloc()` function returns the base mailbox ID if the mailboxes are successfully created. Otherwise, it returns 0.

The maximum number of mailboxes that can be created is determined by a runtime parameter. See Section 18.5, "Running Vera with HDLs."

6.5.3 mailbox_put()

The `mailbox_put()` system task sends data to the mailbox. The syntax is:

```
mailbox_put(mailbox_id, data);
```

mailbox_id - The *mailbox_id* specifies which mailbox receives the data.

data - The *data* can be any general expression that evaluates to a scalar. This includes all data types discussed in Section 3.2, "Data Types and Variable Declaration."

The `mailbox_put()` system task stores data in a mailbox in a FIFO manner.

Note – `mailbox_put()` can ONLY be used with `mailbox_get()`. Using `mailbox_put()` with `mailbox_receive()` results in a runtime error.

6.5.4 mailbox_get()

The `mailbox_get()` system function returns data stored in a mailbox. The syntax is:

```
mailbox_get(wait_option, mailbox_id [, dest_var [, check_option] ]);
```

wait_option - The wait option can either be `NO_WAIT` or `WAIT`. The `NO_WAIT` option continues code execution if the mailbox is empty. The `WAIT` option suspends the process until a message is sent to the mailbox.

mailbox_id - The *mailbox_id* specifies which mailbox data is being retrieved from.

dest_var - The *dest_var* is the destination variable of the mailbox data.

check_option - The *check_option* is an optional argument that should be set to **CHECK** when used. It specifies whether type checking occurs between the mailbox data and the destination variable.

The `mailbox_get()` system function assigns any data stored in the mailbox to the destination variable and returns the number of entries in the mailbox, including the entry just received. If there is a type mismatch between the data sent to the mailbox and the destination variable, a runtime error occurs unless the **CHECK** option is used. If the **CHECK** option is active, a -1 is returned, and the message is left in the mailbox and is dequeued on the next `mailbox_get()` function call. If the mailbox is empty, the function waits for a message to be sent, depending on the wait option. If the wait option is **NO_WAIT**, the function returns a 0.

If no destination variable is specified, the function returns the number of entries in the mailbox, but it does not dequeue an item from the mailbox.

For example, this can be used to continue generating mailbox entries until a specified number are generated:

```
mboxID=alloc(MAILBOX, 0, 1);
while (mailbox_count <11)
{
    mb_data=random();
    mailbox_put(mboxID, mb_data);
    mailbox_count=mailbox_get(NO_WAIT, mboxID);
}
```

This example generates random numbers and puts them in the mailbox. The loop continues while the number of entries is less than 11.

The mailbox waiting queue is FIFO based. By default, a process will wait at a mailbox without timing out. Users can set a time limit with the `timeout()` system task. See Section 6.5.5, "Mailbox Example."

6.5.5 Mailbox Example

This is an example of basic mailbox usage:

```
mboxID=alloc(MAILBOX, 0, 1);
fork
{ repeat(256)
    { randomVar=random();
      address()=randVar[17:0];
      @1 memsys.request[0]=1'b1;
      @2,20 memsys.grant==2'b01;
      data0=randVar[7:0];
      writeOp(address0, data0);
      mailbox_put(mboxID, {address0, data0});
      @1 memsys.request[0]=1'b0;
      @2,20 memsys.grant==2'b00;
```

```

        random_wait();
    }
}
{ repeat(256)
  { mb_var=mailbox_get(WAIT, mboxID, message, CHECK);
    address1=message[15:8];
    data1=message[7:0];
    @1 memsys.request[1]=1'b1;
    @2,20 memsys.grant==2'b10;
    readOp(address1,data1);
    @1 memsys.request[1]=1'b0;
    @2,20 memsys.grant==2'b00;
    random_wait();
  }
}
join

```

This example allocates a mailbox before the fork. The first thread then randomly assigns values to *address0* and *data0*. The data is then passed through a mailbox to the second thread, which is waiting for the data. That data is read into *message* and used for the *readOp* call.

6.6 Timeout Limit

A process will wait forever in semaphore, region and mailbox if the waiting resources are not available. However, the system task *timeout()* can be used to set a time limit. The syntax is:

```
timeout(object_type, cycle_limit [, object_id]);
```

object_type - The *object_type* specifies the type of object for which the timeout is defined. It must be **EVENT**, **SEMAPHORE**, **REGION**, or **MAILBOX**.

cycle_limit - The *cycle_limit* specifies the maximum number of cycles any request will wait.

object_id - The *object_id* specifies an individual resource for which the timeout is set. If it is not specified, the timeout exists for all objects of the type specified.

When the *timeout()* system task is used, it sets the maximum number of cycles that an object will wait for a request. The cycles are based on the *SystemClock*. If the cycle limit is set to 0 cycles, the timeout is disabled. You can specify a timeout for a specific event or for all events of a certain type.

When a semaphore, region, or event times out, a verification occurs.

These are examples of timeout statements:

```

timeout(SEMAPHORE, 100);
timeout(REGION, 50, r_ID);
timeout(EVENT, 20);
timeout(myevent, 300);

```

Note – Specific timeouts take precedence over global timeouts.

6.7 Backward Compatibility

Versions of Vera prior to Vera 4.0 used different commands for mailbox usage: `mailbox_send()` and `mailbox_receive()`.

6.7.1 `mailbox_send()`

The `mailbox_send()` system task sends data to the mailbox. The syntax is:

```
mailbox_send(mailbox_id, data);
```

mailbox_id - The *mailbox_id* specifies which mailbox receives the data.

data - The *data* can be an integer or bit vector of any size.

The `mailbox_send()` system task stores data in a mailbox in a FIFO manner.

Note – `mailbox_send()` can be used with either `mailbox_receive()` or `mailbox_get()`.

6.7.2 `mailbox_receive()`

The `mailbox_receive()` system function returns data stored in a mailbox. The syntax is:

```
mailbox_receive(wait_option, mailbox_id);
```

wait_option - The wait option can either be `NO_WAIT` or `WAIT`. The `NO_WAIT` option continues code execution if the mailbox is empty. The `WAIT` option suspends the process until a message is sent to the mailbox.

mailbox_id - The *mailbox_id* specifies which mailbox data is being retrieved from.

The `mailbox_receive()` system function returns the data stored in the mailbox if there is any. If the mailbox is empty, the function waits for a message to be sent, depending on the wait option. If the wait option is `NO_WAIT`, the function returns a 0 and sets the error flag (see Section 7.3.1, "Error Handling" for information on error flags).

The mailbox waiting queue is FIFO based. By default, a process will wait at a mailbox without timing out. Users can set a time limit with the `timeout()` system task. See Section 6.5.5, "Mailbox Example."

Example 2: Flow Control

This is an example of the arbiter in Example 1. The example includes these features:

- Fork and join
- Triggers
- Semaphores
- Regions
- Mailboxes

This example includes these files:

- *arb.if.vr*
- *signal_drive.vr*
- *RUN* (a run script)

arb.if.vrh

```
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE

interface arb
{
    output [31:0] mem_addr PHOLD OUTPUT_SKEW;
    inout [15:0] mem_data INPUT_EDGE PRZ OUTPUT_SKEW vca rz; // vca monitor
    output oe PHOLD OUTPUT_SKEW;
    output ce PHOLD OUTPUT_SKEW;
    input ack INPUT_EDGE vca r0; // vca monitor
    input clk CLOCK;
}

port generic_port
{
    addr;
    data;
    oe;
    ce;
    ack;
}

bind generic_port port_a
{
    addr arb.mem_addr;
```

```

    data arb.mem_data;
    oe arb.oe;
    ce arb.ce;
    ack arb.ack;
}

bind generic_port port_b
{
    addr arb.mem_addr;
    data arb.mem_data;
    oe arb.oe;
    ce arb.ce;
    ack arb.ack;
}

```

signal_drive.vr

```

#include <vera_defines.vrh>
#include "arb.if.vrh"
#define RSEED 32'h1234_5678
#define MB_ID_A 0
#define MB_ID_B 1

// modified from ch4_ex to use semaphore as an arbiter
// to directly access the memory

// port_a and port_b are used to access write-read sequences
// region_enter/region_exit is deployed to protect memory
// address being used by a either port not to be corrupted
// by the other

// mailbox are use to pass address information from the write
// process to read process

// trigger and sync are used for illustrative to signal the
// end of processes

program signal_test
{
    integer sem_id; // global semaphore
    integer reg_id; // global region
    integer mbox_id[2]; // global mailbox
    integer randseed = RSEED;
    event evt[4]; // global events

```

```

// reset the oe and ce
@0 arb.ce <= 1'b0 ;
@0 arb.oe = 1'b0 ;

// turn vca on for unexpected signal changes
vca ( ON, arb );

// allocate 1 semaphore with initial value of 1
sem_id = alloc ( SEMAPHORE, 0, 1, 1 );
if ( sem_id == 0 ) error ("semaphore allocation failed!");

// allocate 1 region
reg_id = alloc ( REGION, 0, 1 );
if ( reg_id == 0 ) error ("region allocation failed!");

// initiate burst write-read with both port_b and port_a at the same time
fork
{
// port b
integer i;
bit [31:0] addr;
bit [15:0] data[4];

repeat (10)
{
addr = random(randseed) & 32'h0000_ffff;
region_enter ( WAIT, reg_id, addr ); // protect the addr
for ( i = 0; i < 4; i++ )
data[i] = random (randseed); // generate data
do_write with port_b ( addr, data ); // do actual write

// pass the addr and data to the read-process
mailbox_put ( mbox_id[MB_ID_B], addr );
mailbox_put ( mbox_id[MB_ID_B], data[0] );
mailbox_put ( mbox_id[MB_ID_B], data[1] );
mailbox_put ( mbox_id[MB_ID_B], data[2] );
mailbox_put ( mbox_id[MB_ID_B], data[3] );
} //end repeat
trigger ( ONE_BLAST, evt[0] ); // signal finish
}
{
// port_a
integer i;
bit [31:0] addr;
bit [15:0] exp_data[4], data[4];

```

```

repeat ( 10 )
{
    mailbox_get ( WAIT, mbox_id[MB_ID_B], addr ); //capture the addr
    do_read with port_b ( addr, data ); // do actual read

    // verify the data read

    for ( i = 0; i < 4; i++ )
    {
        // capture the expected data value
        mailbox_get ( WAIT, mbox_id[MB_ID_B], exp_data[i] );
        if ( data[i] != exp_data[i] )
            error ( "port_b transactions gives inconsistent data!\n");
        else
            printf ("cycle %0d: port_b data %0d OK! \n", get_cycle(),i );
    }
    region_exit ( reg_id, addr ); //done with the addr
} //end repeat
trigger ( ONE_BLAST, evt[1] ); // signal finish
}

{
// port a
integer i;
bit [31:0] addr;
bit [15:0] data[4];

repeat (10)
{
    addr = random(randseed) & 32'h0000_ffff;
    region_enter ( WAIT, reg_id, addr ); // protect the addr
    for ( i = 0; i < 4; i++ )
        data[i] = random (randseed); // generate data
    do_write with port_a ( addr, data ); // do actual write

    // pass the addr and data to the read-process

    mailbox_put ( mbox_id[MB_ID_A], addr );
    mailbox_put ( mbox_id[MB_ID_A], data[0] );
    mailbox_put ( mbox_id[MB_ID_A], data[1] );
    mailbox_put ( mbox_id[MB_ID_A], data[2] );
    mailbox_put ( mbox_id[MB_ID_A], data[3] );
} //end repeat
trigger ( ONE_BLAST, evt[2] ); //signal finish
}

{
// port_a

```

```

integer i;
bit [31:0] addr;
bit [15:0] exp_data[4], data[4];

repeat ( 10 )
{
    mailbox_get (WAIT, mbox_id[MB_ID_A], addr); // capture the addr
    do_read with port_a ( addr, data ); // do actual read

    // verify the data read
    for ( i = 0; i < 4; i++ )
    {
        // capture the data
        mailbox_get ( WAIT, mbox_id[MB_ID_A], exp_data[i] );
        if ( data[i] !== exp_data[i] )
            error ( "port_a transactions gives inconsistent data!\n" );
        else
            printf ("cycle %0d: port_a data %0d OK!\n",get_cycle(),i);
    }
    region_exit ( reg_id, addr ); // release the addr
} //end repeat
trigger ( ONE_BLAST, evt[3] ); // signal finish
}

join none
sync ( ALL, evt[0], evt[1], evt[2], evt[3] );
printf ("\n----- All Finished ----- \n");
} //end program

// do_read : acquire semaphore and do burst read at the addr
// release semaphore by the end

task do_read with generic_port ( bit [31:0] addr, var bit [15:0] data[4] )
{
    // acquiring lock on to driving/sampling signals
    semaphore_get ( WAIT, sem_id, 1 );
    @1 $addr <= addr; // drive the addr, oe, ce bits
    $oe = 1'b1;
    $ce = 1'b1;
    @2, 10 $ack == 1'b1; // ack should have come within 8 cycles
    fork
    {
        @0, 3 $ack == 1'b1; // ack should stay for 4 cycles
        @0 $oe <= 1'b0; // at the end, deassert oe and ce
        @0 $ce = 1'b0;
    }
}

```

```

        integer i;
        for (i = 0; i < 4; i++) // latch the data for 4 consecutive cycles
        {
            @(posedge CLOCK);
            data[i] = $data;
        } //end for
    }
    join all
    @1 $oe = void; // void drive to kill one more cycle

// releasing lock

    semaphore_put ( sem_id, 1 );
} //end do_read

// do_write : acquire semaphore and do burst write at the addr
// release the semaphore by the end

task do_write with generic_port ( bit [31:0] addr, bit [15:0] data[4] )
{
    // acquiring lock on to driving/sampling signals
    semaphore_get ( WAIT, sem_id, 1 );
    @0 $addr <= addr; // drive the addr, oe, ce bits
    $oe = 1'b0;
    $ce = 1'b1;
    @2, 10 $ack == 1'b1; // ack should have come within 8 cycles
    fork
    {
        @0, 3 $ack == 1'b1; // ack should stay for 4 cycles
        @0 $ce = 1'b0;
    }
    {
        integer i;
        for (i = 0; i < 4; i++) // drive the data for 4 consecutive cycles
        {
            $data = data[i] async;
            @(posedge CLOCK);
        } //end for
    }
    join all
    @1 $oe = void; // void drive to kill one more cycle

// releasing lock

    semaphore_put ( sem_id, 1 );
} //end do_write

```

RUN

```
#!/bin/csh -f
rm -f core *.vro *.vshell >& /dev/null
vera -cmp -g signal_drive.vr
```

Copyright © 2000 Synopsys, Inc.

7. Predefined Vera Tasks

Vera provides many predefined system tasks that are immediately available in the Vera environment. This chapter discusses many of the system tasks and details their use. This chapter includes these sections:

- Vera I/O
- Random Number Generators
- Simulation Errors
- Simulation Control
- Reading Plus Arguments

7.1 Vera I/O

The tasks in this section are Vera's input/output tasks.

7.1.1 Output

`printf()`

Vera supports a C-style `printf()` system task that sends information to *stdout* (verilog.log) during the course of a simulation. The syntax is:

```
printf("fmt_str", arg1, arg2, ..., argN);
```

fmt_str - This is a C-style format string. The format specifiers that can be used are:

`%h` or `%H` : print in hexadecimal format

`%d` or `%D` : print in decimal format

`%o` or `%O` : print in octal format

`%b` or `%B` : print in binary format

`%c` or `%C` : print in ASCII character format

`%s` or `%S` : print as string

argN - These are the arguments to be printed.

For `%h`, `%d`, `%o`, `%b`, the values are sized automatically to the maximum possible space needed for the given expression. You can minimize the size by inserting a zero between the `%` character and the letter that indicates the radix. For example:

```
printf("Data = %0h Addr = %0h\n", data, addr);
```

With `%s`, you can also print `bind_var` variables. For example:

```
bind_var current;
current = get_bind();
printf("Current bind = %s\n", current);
```

This example prints the bind name associated with the `bind_var` `current`.

7.1.2 File IO

Vera provides predefined tasks to read and write external files.

`fopen()`

The `fopen()` system function opens a specified file. The syntax is:

```
fopen ("filename", "type");
```

filename - *filename* specifies the file to be opened.

type - The argument *type* is one of the following:

- `r`: open for reading
- `w`: truncate or create for writing
- `a`: append or create for writing

The `fopen()` function opens the specified file and returns a 32-bit file descriptor or 0 if it fails.

`fclose()`

The `fclose()` system task closes the specified file. The syntax is:

```
fclose (file_descriptor);
```

file_descriptor - There are three predefined file descriptors in *vera_defines.vrh*:

- `stdin`: terminal input
- `stdout`: terminal output
- `stderr`: terminal error output

`fprintf()`

The `fprintf()` system task writes the output to a specified file. The syntax is:

```
fprintf (file_descriptor, fmt_str);
```

fmt_str - The format string is the same as the `printf()` system task.

freadb()

The **freadb()** system function reads binary data from a specified file and returns it as a bit vector. The syntax is:

```
freadb(file_descriptor);
```

The **freadb()** function reads binary formatted data from a specified text file, line by line, and returns the data as a bit vector. Lines with only white spaces and comments are ignored. Each line of the file must be in this format:

```
white_space* binary comment*
```

white_space - The **white_space** can be any number of spaces.

binary - The **binary** must be a combination of '0', '1', 'z', 'x', 'Z', 'X' and '_'.

comment - The **comment** must be a Vera comment starting with "//".

When the end of the file is reached, the function sets the error flag. For more information on the error flag, see Section 7.3, "Simulation Errors."

freadh()

The **freadh()** system function reads hexadecimal data from a specified file and returns it as a bit vector. The syntax is:

```
freadh(file_descriptor);
```

The **freadh()** function reads hex formatted data from a specified text file, line by line, and returns the data as a bit vector. Lines with only white spaces and comments are ignored. Each line of the file must be in this format:

```
white_space* hex comment*
```

hex - The **hex** must be a combination of '0', '1', 'a' to 'f', 'A' to 'F', 'x', 'z', 'X', 'Z' and '_'. The **comment** must be a Vera comment starting with "//".

When the end of the file is reached, the function sets the error flag. For more information on the error flag, see Section 7.3, "Simulation Errors."

freadstr()

The **freadstr()** system function returns a string from a specified file. The syntax is:

```
freadstr(file_descriptor, mode);
```

mode - The **mode** can be **VERBOSE**, which prints a warning and returns a null string when the end of the file is reached, **SILENT**, which returns a null string when the end of the file is reached, or **RAWIN**, which is the same as **SILENT** except that comments and blank lines are not filtered out. If the **mode** is not specified, the default is **VERBOSE**.

The `freadstr()` function returns a string containing a line of text from a specified file. The returned string does not contain line-feed characters. Comments and blank lines in the input file are ignored by the function, unless the **RAWIN** mode is used.

When the end of the file is reached, the function sets the error flag. For more information on the error flag, see Section 7.3, "Simulation Errors."

rewind()

The `rewind()` system task moves the file access pointer to the beginning of the file. The syntax is:

```
rewind(file_descriptor);
```

File I/O Example

This is an example that incorporates many of the predefined I/O functions:

```
#include <vera_defines.vrh>

program file_io_test
{
    // Read 10 lines of 16-bit hex data from an input file
    // and write it to another file
    bit [15:0] in_data[], out_data[];
    string in_data_str;
    integer fdi, fdo, i;

    fdi = fopen ( "input.dat", "r" );
    if ( fdi == 0 )
        error ("Can't open input.dat!\n");

    // input method 1 - freadh
    for ( i = 0; i < 10; i++ )
        in_data[i] = freadh ( fdi );
    rewind ( fdi );

    // input method 2 - freadstr
    // and output method 1 - fprintf

    fdo = fopen ( "output.dat", "w" );
    if ( fdo == 0 )
        error ("Can't open output.dat!\n");
    i = 0;
    in_data_str = freadstr ( fdi );
    while ( in_data_str.len() > 0 )
    {
        sscanf ( in_data_str, "%h", out_data[i] );
        if ( out_data[i] != in_data[i] )
```

```

        error ("Something's wrong\n");
    else
        fprintf ( fdo, "Output Data : %h\n", out_data[i] );
        i++;
        in_data_str = freadstr ( fdi );
    }
}

```

7.2 Random Number Generators

Vera provides pseudo-random number generators, which can be used for writing random tests and for randomizing test data. The random number generator is based on the Unix random number generator, which uses a non-linear, additive feedback algorithm with a 256-byte state and a period of at least.

random()

```
random([seed]);
```

seed - The *seed* is an optional argument that determines which random number is generated. The *seed* can be any valid Vera expression, including variable expressions. The random number generator generates the same number every time the same seed is used.

The random number generator is based on the Unix random number generator, which uses a non-linear additive feedback algorithm with a 256-byte state and a period of at least 2^{35} . For more information on the Unix random number generator, see the Unix man pages (from a Unix shell, type "man random").

The random number generator is deterministic. Each time you restart your program, it cycles through the same random sequence. You can make this sequence indeterminate by seeding the **random()** function with an extrinsic random variable, such as the time of day.

To generate random numbers, first call the **random()** system function with a 32-bit seed value. Then call the **random()** function without a seed each time you need a new random number. For example:

```

random( 184984 ); // Initialize the generator
addr = {random(),random()};
ph_number = random() >> 5;

```

urandom()

The **urandom()** function generates an unsigned 32-bit random number based on the same algorithm used by the **random()** function. The syntax is:

```
urandom([seed]);
```

rand48()

The **rand48()** function generates a 32-bit non-negative random number based on the *lrand48()* algorithm. The syntax is:

```
rand48([seed]);
```

For more information on the *lrand48()* algorithm, see the Unix man pages (from a Unix shell, type "man lrand48").

urand48()

The **urand48()** function generates an unsigned 32-bit random number based on the *mrnd48* algorithm. The syntax is:

```
urand48([seed]);
```

For more information on the *mrnd48()* algorithm, see the Unix man pages (from a Unix shell, type "man mrnd48").

7.3 Simulation Errors

Vera provides a set of predefined tasks and functions that are used in handling simulation errors and in assisting the debugging process.

7.3.1 Error Handling

error()

The **error()** system task generates a Vera simulation error. The syntax is:

```
error("fmt_str");
```

fmt_str - The *fmt_str* is of the same form as the **printf()** function.

You can intentionally generate a simulation error condition with the **error()** system task. The system task prints the message specified with *fmt_str* and then generates the error.

For example:

```
if( length > 256 )
  error( "Too long length %d specified\n", length );
```

flag()

The **flag()** system function sets and clears error flags. The syntax is:

```
flag([value]);
```

value - The *value* flag determines how the error flag is set. The *value* can be **ON**, which sets the error flag, or **OFF**, which clears the error flag.

The `flag()` function sets and clears error flags that are raised when various non-fatal simulation errors occur (such as soft-expects and semaphore timeouts). The function returns the value of the flag before setting or clearing the flag. If no argument is passed in the call, the function only returns the state of the error flag.

If the error flag is set in a child process (in a `fork/join` block), the error flag is transferred to its parent only if the parent is waiting for the child. Returning from a task or procedure with the flag raised triggers a simulation error (default setup), and the flag is cleared.

For example:

```
foo_bus.data == 8'h54 soft; // soft expect
if( flag () ) {
    printf( "Warning: data is not 8'h54\n" );
    flag( OFF );
    ...
}
```

`error_mode()`

The `error_mode()` system task sets Vera's error generation mode. The syntax is:

```
error_mode(type, error_class);
```

type - The *type* must either be **ON**, which activates the error generation mode, or **OFF**, which deactivates the error generation mode.

error_class - The *error_class* and its defaults are listed in Table 7-1.

Table 7-1 error_class definition

Error Class	Definition	Default
EC_ARRAYX	Error on indexing array with X value	on
EC_EXPECT	Error on Expect fail ("@")	on
EC_FULLEXPECT	Error on Full-Expect fail ("@@")	on
EC_SEXPECT	Error on Soft-Expect fail ("@"soft)	off
EC_SFULLEXPECT	Error on Soft-Full-Expect fail ("@@"soft)	off
EC_RETURN	Return from subroutines with error flag set, issues error message, and clears error flag	on
EC_USERSET	flag(ON) system task is called by user	off
EC_SEMTMOUT	semaphore time out error	on
EC_RGNTMOUT	region timeout error	on
EC_MBXTMOUT	mailbox timeout error	on
EC_CONFLICT	drive conflict error	on
EC_SCONFLICT	Soft drive conflict error	off

For example:

```
error_mode( ON, EC_USERSET );
error_mode( OFF, EC_RETURN );
```

7.3.2 Debug Support Routines

trace()

The `trace()` system task enables and disables trace message generation for various types of objects. The syntax is:

```
trace(request, object [, id_level]);
```

request - The *request* must either be **ON**, which activates trace message generation, or **OFF**, which deactivates trace message generation.

object - The *object* argument definitions are listed in Table 7-2.

Table 7-2 Trace object definitions

Object	Action
SEMAPHORE	trace message for semaphore
REGION	trace message for region
MAILBOX	trace message for mailbox
PROGRAM	trace message for program
VERBOSE	verbose message
<event_variable>	trace all events on <event_variable>

id_level - If the *object* argument is an event variable or **VERBOSE**, the *id_level* is not necessary.

If the *object* argument is **SEMAPHORE**, **REGION**, or **MAILBOX**, the *id_level* specifies the name of traced object.

If the *object* argument is **PROGRAM**, *id_level* specifies the trace level. Level 0 only traces the current process. Level 1 generates traces on child processes as well. For trace levels greater than 2, the trace level is set to 1 for its subroutines. For example, with `trace (ON, PROGRAM, 3)`, the trace level becomes 2 at the subroutine.

Note – A new trace call overrides the given trace level.

If the *object* argument is **VERBOSE**, the `trace()` function controls the generation of the verbose messages, including the warning messages on soft-expect fail, etc.

When the program trace is enabled, the Vera-VS generates a trace dump along with the internal execution. For example:

```
<TRACE> Context(0): program rarb_test 22 in rarb.vr
<TRACE> Context(0): program rarb_test 23 in rarb.vr
<TRACE> Context(0): program rarb_test 24 in rarb.vr
```

The trace shows the context ID (unique ID allocated by the system), program or subroutine name, line number, and file name.

Program Trace Examples

Here are examples of program traces:

```
trace( ON, PROGRAM, 5 );
trace( ON, SEMAPHORE, sem_id );
trace( ON, REGION, addr_begin );
trace( ON, MAILBOX, ups_mbox );
trace( ON, all_done_evt );
trace( ON, VERBOSE );
trace( OFF, PROGRAM, 1 );
```

7.4 Simulation Control

Vera provides a set of tasks and functions used to control simulations.

stop()

The `stop()` system task stops the simulation when it is encountered. The syntax is:

```
stop();
```

The `stop()` system task is equivalent to the Verilog task `$stop`. If you are running a Verilog simulation, the simulation stops, reports that a stop has been encountered, and exits to a Verilog prompt when the task is encountered. Normal Verilog commands can be issued at the command line. To continue the simulation, enter a period (.) at the command line.

If you are running Vera-CS standalone, the simulation stops, reports that a stop has been encountered, and prompts you to hit RETURN to continue.

exit()

The `exit()` system task exits the simulation when it is encountered. The syntax is:

```
exit(status);
```

status - The *status* must be an integer constant or integer variable. Its value is assigned to the environment variable `$status` when the simulation is exited.

The `exit()` exits the simulation when it is encountered. It assigns the *status* value to the environment variable *\$status*, which can be evaluated from the Unix prompt. It is particularly useful for evaluating flag and variable values when the simulation exits.

`get_cycle()`

The `get_cycle` system function returns the current simulation cycle count. The syntax is:

```
get_cycle();
```

Vera counts the internal simulation cycles at the positive edge of `SystemClock`. The system function `get_cycle()` returns the current simulation cycle as a 32-bit unsigned value.

For example:

```
printf( "Current Cycle: %d \n", get_cycle() );
```

`get_time()`

The `get_time` system function returns the current 64-bit simulation time as two 32-bit values. The syntax is:

```
get_time(word);
```

word - The *word* must be either `LO`, which returns the lower 32-bit value, or `HI`, which returns the higher 32-bit value.

The simulation time is evaluated from the HDL side, or from the Vera side when running Vera-CS standalone.

For example:

```
printf( "Current Time: %d.%d \n", get_time(HI), get_time(LO) );
```

`get_systime()`

The `get_systime()` system function returns the number of seconds since 00:00: UTC, January 1, 1970. The syntax is:

```
get_systime();
```

The `get_systime()` system function corresponds to the Unix library function `time()` (see the Unix man pages for more information).

7.5 Reading Plus Arguments

Vera can read Verilog plus arguments using the `get_plus_arg()` system function. The VHDL equivalent to plus arguments is included in the *vera.ini* file, discussed in Section 17.3.2, "VHDL Plus Arguments."

get_plus_arg()

Within a Vera program you can check any plus arguments on the Verilog command line by using the `get_plus_arg()` system task.

```
get_plus_arg(request, plus_arg);
```

request - The valid values for *request* are listed in Table 7-3.

Table 7-3 Plus Argument Requests

Request	Action
CHECK	returns 1 if the specified plus argument is present
HNUM	returns a hexadecimal number attached to the specified plus argument
NUM	returns an integer attached to the specified plus argument
STR	returns a string attached to the specified plus argument (returns a bit string type and not a string primitive type)

plus_arg - The *plus_arg* is the plus argument you want to evaluate.

The `get_plus_arg()` system function returns a value based on the *request* type and *plus_arg* value.

For example:

```
#include <vera_defines.vrh>
#define DEF_RP_TIMES 10
#define DEF_RSEED 32'habcd_ef01
#define DEF_LOG_FILE_NAME "test.log"

program test
{
    integer repeat_times = DEF_RP_TIMES;
    bit [31:0] random_seed = DEF_RSEED;
    bit [2047:0] bit_str; //max 256 char
    string log_file_name = DEF_LOG_FILE_NAME;

    // get repeat times if any
    if ( get_plus_arg ( CHECK, "set_repeat_times=" ) )
    {
        repeat_times = get_plus_arg ( NUM, "set_repeat_times=");
    }

    // get random seed if any
```

```

    if ( get_plus_arg ( CHECK, "set_random_seed=" ) )
    {
        random_seed = get_plus_arg ( HNUM, "set_random_seed=" );
    }

    // get log file name if any

    if ( get_plus_arg ( CHECK, "set_log_file_name=" ) )
    {
        bit_str = get_plus_arg ( STR, "set_log_file_name=" );
        log_file_name.bittostr ( bit_str );
    }

    printf ( " repeat times is %0d\n", repeat_times );
    printf ( " random seed is %h\n", random_seed );
    printf ( " log file name is %s\n", log_file_name );
}

```

This example is invoked with this command line:

```

vera_cs filename.vro +set_repeat_times=7 +set_log_file_name=log.file \
    +set_random_seed=3

```

8. Object-Oriented Programming

Object-oriented programming is at the very core of Vera. The implementation is clear, straightforward and elegant, assimilating and extending the best of the features found in C++ and Java. For the new user, this implementation will prove easy to learn and powerful to use. The advanced user will find the Vera implementation to be intellectually deep and compelling.

Most of the features of Vera are available without using the object-oriented framework. You can be a good Vera programmer without ever defining a class, instantiating an object, or invoking a method. The basic object-oriented capabilities, however, are straightforward and easy to learn, and the consequent improvements in your productivity will be immense.

By using the object-oriented paradigm, your program will be easier to design, easier to develop, easier to debug, easier to maintain, easier to share, and easier to re-use. This all follows from the fundamental orientation of the paradigm, which leads you to group related data and code into separable and separated units called classes. A class might be as low-level as a packet or as high-level as a complete processor description. Each instance of a class, called an *object*, has its own copy of its data as well as access to the code that acts on this data. Classes have both public and private data and code. Private data and code inside the class is inviolate - outside programs cannot access it, except indirectly through public access methods. Almost as importantly, code inside the class can't have any unforeseen side-effects outside of the class. The focus of all code actions is strictly inside the class.

Object-oriented programming leads naturally to grouping related code and data together, keeping these sections small, easy to understand, easy to debug and easy to maintain. It also leads to crisply formalizing the interactions among these classes. The result is a discipline and program structure that is useful for small programs, but has a dramatic, powerful impact on your productivity as soon as your program becomes large or in any way complex.

Object-oriented programming, especially with a clean implementation such as is found here, is more a matter of perspective and mindset than of features and syntax. The paradigm is built on three tenets: encapsulation, inheritance and polymorphism.

Encapsulation is the concept of bringing related code and data together and making access into or out of the class formal and rigorous, hiding almost all of the detail in the class from the outside world.

Inheritance is the idea that once a class is debugged, you don't want to modify it for a new application or use. You would rather inherit features you can use and replace features you need to change. This leads to the notion of a base class with general, widely applicable core features and a class hierarchy of subclasses, where each member of the hierarchy inherits core features from its parent, adds new features that make the subclass more specific and refined, and then

passes this more complete set on to its descendants. An example might be a base class that defines vehicles - providing the basic properties that are generic to all vehicles. We could then extend these properties to include the properties specific to land vehicles. This could be further extended to give us wheeled vehicles, then cars, then Toyotas and ultimately the Toyota Tercel - an actual example of a vehicle, a land-vehicle, a wheeled vehicle, etc. Later, we could derive a new class for air-vehicles, motorcycles or Ferraris - all by extending the appropriate subclass. If we planned carefully we would not have to redo a lot of work, and the extensions would be straightforward.

The third tenet of the object-oriented paradigm is polymorphism. This allows us to wait until run time to bind object variables to their data type. Traditional languages with simple data types don't allow you to do this; an integer is different from a string, and the two could never occupy the same variable. But with object-oriented programming, the range of data types are unlimited, and this traditional restriction often is a hindrance. For example, we might have a variable called `coordinate` where we have different versions (*classes*) defining a coordinate: cartesian, polar, spherical, cylindrical, in two or three dimensions. The subroutines would have the same names in all of the class declarations; all might have a subroutine that could compute the distance between two points or the distance from the origin to a point, `distance(coordinate c)`. With polymorphism, when you call the subroutine `distance()` at run time, the system looks at the type of the data (of `coordinate`) and decides which version of `distance` to invoke.

8.1 Classes and Objects

A *class* is a collection of data and a set of subroutines that operate on that data. A class's data is referred to as *properties*, subroutines are called *methods*, and we will refer to both as *members* of the class. The properties and methods, taken together, usually define the contents and capabilities of some kind of *object*.

For example, a packet is an object. It might have a command field, an address, a sequence number, a time stamp, and a packet payload. In addition, there are various things we can do with a packet: initializing the packet, setting the command, reading the packet's status, checking the sequence number. Each Packet is different, but as a *class*, packets have certain intrinsic properties that we can capture in a definition.

Example:

```
class Packet {

    bit [3:0] command;      // data portion
    bit [40:0] address;
    bit [4:0] master_id;
    integer time_requested;
    local integer time_issued;
    local integer status;

    task new() {            // initialization
        command = IDLE;
```

```

        address = 41'b0;
        master_id = 5'bx;
    }

    /* private operations, internal to the class */
    local task clean() {
        command = 0; address = 0; master_id = 5'bx;
    }

    /* public access entry points */
    task issue_request( integer delay ) {
        // send request to bus
        // ...
    }
    function integer current_status(){
        current_status = status;
    }
}

```

Note that a common convention is to capitalize the first letter of the class name, so that it is easy to recognize class declarations.

8.2 Objects and Instance of Classes

So far, we only have the definition of the *class* Packet. We have created a new, complex data type but we can't do anything with the class itself. We need to create an *instance* of the class, a single Packet object. The first step is to create a variable that can hold an object's name (or handle):

```
Packet p;
```

Nothing has been created yet. We have just declared that p is a variable that can hold the handle of a Packet object. In Vera, for p to refer to something, we need to explicitly create an instance of the class using the new keyword.

```
Packet p;
p = new;
```

You can detect uninitialized object handles by comparing them with null. For example:

```

class obj_foo
{
    ...
}

task mytask (integer a, (obj_foo myfoo = null))
{
    if (myfoo == null) myfoo = new;
}

```

This example checks if myfoo is initialized. If it is not, it initializes it with the new command.

8.3 Accessing Object Properties

Now that we have created an object, we can use its data fields by qualifying property names with an instance name. Looking at the earlier example, we can use the commands for our Packet *p* as follows:

```
Packet p = new;
p.command = INIT;
p.address = random();
time = p.time_requested;
```

8.4 Using Object Methods

To access an object's methods, we use the same syntax we used to access properties:

```
Packet p = new;
status = p.current_status();
```

Note that we did not say

```
status = current_status(p);
```

The focus in object-oriented programming is the object, in this case the packet, not the function call. Also, objects are self-contained, with their own *methods* for manipulating their own properties. So we don't have to pass arguments to `get_status()`. The properties of a class are freely and broadly available to the methods of the class, but each method only accesses the properties associated with its object, its instance.

8.5 Constructors

Vera does not require the complex memory allocation and de-allocation of C++. Construction of an object is straightforward and garbage collection, as in Java, is implicit and automatic. There can be no memory leaks or other subtle behavior that is so often the bane of C++ programmers.

Vera provides a mechanism for initializing an instance at the time the object is created. When you create an object, for example

```
Packet p = new;
```

the system executes the new task associated with the class:

```
class Packet {
    integer command;
    task new () { command = IDLE; }
```


Note that *new* is now being used in two very different contexts with very different semantics. The variable declaration creates an object of class *Packet*. In the course of creating this instance, the *new* subroutine is invoked, if it exists, allowing you to do any initialization or start-up functions you require. The *new* task is also called the *constructor* of a class.

It is also possible to pass arguments to the constructor, to allow for run-time customization of the object:

```
Packet p = new(STARTUP, random(), get_time(LO));
```

where the *new* initialization task in *Packet* might now look like

```
task new (integer in_command=IDLE, bit[40:0] in_address=0,
          integer time_stamp=0) {
    command = in_command; address = in_address;
    time_requested = time_stamp;
}
```

The conventions for arguments are the same as for subroutine calls, including the use of default arguments.

8.6 Class Properties

So far, we have only declared *instance* properties. Each instance of the class, each *Packet*, has its own copy of each of its three variables. There are also cases where we only want one copy of the variable, to be shared by all instances. These *class properties* are created using the *static* keyword. Thus, for example, in a case where all instances of a class need access to a semaphore id, we might have

```
class Packet {
    static integer semId = alloc(SEMAPHORE, 0, 1, 1);
```

Now, *semId* will be created and initialized the first time an object of the *Packet* class is created. Thereafter, every packet object can access the semaphore in the usual way:

```
Packet p;
semaphore_get(WAIT, p.semId);
```

8.7 this

There are times when you need to unambiguously refer to properties or methods in the current instance. For example, the following declaration is a common, clean way to write an initialization routine;

```

class Demo {
    integer x;
    task new (integer x) {
        this.x = x;
    }
}

```

`x` is now both a property of the class and an argument to the task `new`. In the task `new`, an unqualified reference to `x` will be resolved by looking at the innermost scope, in this case the subroutine argument declaration. To access the instance property, we qualify it with `'this'` to refer to the current instance.

Note that in writing methods, you can always qualify members with `this` to refer to the current instance, but it is usually unnecessary.

8.8 Assignment, Re-naming and Copying

When we declare a class variable, we have only created a name for an object. Thus

```
Packet p1;
```

creates a variable, `p1`, that can hold the handle of an object of class `Packet`, but the initial value of `p1` is `null`. It is not until we create an instance of type `Packet` that the object exists, and that `p1` contains an actual handle:

```
p1 = new;
```

Thus, if we create another variable

```
Packet p2;
```

and assign `p1` to `p2`

```
p2 = p1;
```

then we still have only one object, which we can refer to with either the name `p1` or `p2`. Note, we have only executed `new` once, so we have only created one object.

If we rewrite the last expression slightly differently, however, we make a copy of `p1`:

```
p2 = new p1;
```

Now we have executed `new` twice, so we have created two objects. With this syntax, however, `p2` will be a copy of `p1`, but it will be what is known as a shallow copy. All of the variables are copied across: integers, strings, instance handles, etc. Objects, however, are not copied, only their handles; as before, we have created two names for the same object. This is true even if the class declaration includes the instantiation operator `new`:

```

class A {integer j = 5; }
class B {
    integer i = 1;
    A a = new;
}

program test {
    integer test;
    B b1 = new;           // Create an object of class B
    B b2 = new b1;        // Create an object that is a copy of b1
    b2.i = 10;            // i is changed in b2, but not in b1
    b2.a.j = 50;          // change object a, shared by both b1 and b2
    test = b1.i;          // test will be set to 1 (b1.i has not changed)
    test = b1.a.j;        // test will be set to 50 (a.j has changed)
}

```

Note several things. We can initialize properties and instantiate objects directly in a class declaration. Second, the shallow copy does not copy objects. Third, we can chain instance qualifications as needed to reach into objects or to reach through objects:

```

b1.a.j    // reaches into a, which is a property of b1

p.next.next.next.next.val // would chain through a sequence of
                           // handles to get to val.

```

To do a full (deep) copy, where everything (including nested objects) are copied, you need to write custom code. Thus, we might have

```

Packet p1 = new;
Packet p2 = new;
p2.copy(p1);

```

where `copy(Packet p)` is a method written to copy the object specified as its argument into its instance.

8.9 Subclasses and Inheritance

We have defined a class that declares a `Packet`. Now, assume we want to extend this declaration so that, for example, we could chain packets together on a list. We could create a new class which has, as a property, a variable of type `Packet`. We would be nesting classes, like a set of Russian dolls; the references, as we saw in the previous example, can quickly become cumbersome.

A much preferred alternative is to extend the class, creating a new subclass that *inherits* the members of the parent class. Thus, for example, we could have

```

class LinkedPacket extends Packet {
    LinkedPacket next;
    function LinkedPacket get_next() { get_next = next; }
}

```

Now, all of the methods and properties of `Packet` are part of `LinkedPacket` - as if they were defined in `LinkedPacket` - and `LinkedPacket` has additional properties and methods.

We can also override the parent's methods, changing their definitions.

8.10 Overridden Members

Subclasses objects are also legal representative objects of their parent classes. For example, every `LinkedPacket` object is a perfectly legal `Packet` object; we can assign the handle of a `LinkedPacket` object to a `Packet` variable:

```

LinkedPacket lp = new;
Packet p = lp;

```

In this case, references to `p` access the methods and properties of the `Packet` class. So, for example, if you have overridden properties and methods in `LinkedPacket`, when you reference these overridden members through `p` you get the original members in the `Packet` class. From `p`, new and overridden members in `LinkedPacket` are hidden from you.

```

class Packet {
    integer i = 1;
    function integer get() { get = i; }
}

class LinkedPacket extends Packet {
    integer i = 2;
    function integer get() { get = -i; }
}

LinkedPacket lp = new;
Packet p = lp;
j = p.i;           // j = 1, not 2
j = p.get();       // j = 1, not -1 or -2

```

Note that this is different from the semantics of, for example, Java. In Java, you would get the original properties, but you would get the overridden methods of the child class. In Vera, to get the overridden method, the parent method needs to be declared `virtual` (see below).

8.11 super Class

Every subclass has a *super* class, its parent class referred to in the *extends* clause of the class declaration. When members have been overridden in the subclass, you can access the super-class versions from methods in the subclass by qualifying the member name with the keyword *super*:

```
super.count;

super.clear_table();
```

This only allows you to reach up one level of the hierarchy; this may be a member declared a level up or a member inherited by the class one level up. There is no way to reach higher (*super.super.count* is not allowed).

(In the sequel, when we refer to subclasses, we will be referring to classes that are extensions of the current class. When we refer to super-classes, these will be the classes that the current class is extended from, beginning with the original base class.)

8.12 Casting

It is always legal to assign subclass variable to a variable of superclass higher in the inheritance tree. It is never legal to directly assign a superclass variable to a variable of one of its subclasses. However, it may be legal to place the contents of the superclass handle in a subclass variable.

- To check if the assignment will be legal, use the function `cast_assign()`:

```
function integer cast_assign(var destination_handle, source_handle);
```

This function checks that the contents of `source_handle` is of class `destination_handle` or one of its subclasses. If it is, `cast_assign` does the assignment; if it is not, `cast_assign()` generates an error and terminates.

A second version of this function allows you to check the results without generating an error:

```
cast_assign(destination_handle, source_handle, CHECK);
```

does the assignment and returns a 1 if the assignment is valid. Otherwise, it sets the destination handle to null and returns a 0.

8.13 Chaining Constructors

When you instantiate a subclass, one of the system's first actions is to invoke the class method `new()`. The first, implicit action `new()` takes is to invoke the `new()` method of its superclass, and so on up the inheritance hierarchy. Thus, all of the constructors get called, in the proper order, beginning with the base class and ending with the current class.

If the initialization method of the super-class requires arguments, you have two choices. If you want to always supply the same arguments, you can specify them at the time you extend the class:

```
class EtherPacket extends Packet(5) {
```

This will pass 5 to the new routine associated with Packet. A more general approach is to use the super keyword, to call the superclass constructor *as the first executable statement of the constructor*:

```
task new() {
    super.new(5);
```

8.14 Data Hiding and Encapsulation

So far, we have made all of our properties and methods available to the outside world without restriction. However, for most data (and most subroutines) we want to hide them away from the outside world, "seal them away in the capsule" of the class. This keeps other programmers from relying on your specific implementation - so you can safely modify it later - and it also protects against accidental modifications to properties that are internal to the class. When all data becomes hidden - being accessed only by public methods - testing and maintenance of the code becomes much easier.

Unlabeled properties and methods are public, available to anyone who has access to the object's name.

A member identified as `local` is available only to methods inside the class. Further, these `local` members are not visible even to subclasses and cannot be inherited. Of course, non-local methods that access `local` properties or methods can be inherited, and work properly as methods of the subclass.

A protected property or method has all of the characteristics of a `local` member, except that it can be inherited; it is visible to subclasses.

Note that within the class, we can reference a `local` method or property of our class, even if it is in a *different* instance. For example

```
class Packet {
    local integer i;
    function integer compare (Packet other) {
        compare = (this.i == other.i);
    }
```

A strict interpretation of encapsulation might say that `other.i` should not be visible inside of this packet, since it is a `local` property being referenced from outside its instance. Within the same class, however, these references are allowed. In this case, `this.i` will be compared to `other.i` and the result of the logical comparison will be returned.

In summary

- wherever possible, use `local` members. Hide members that the outside world doesn't need to know about;
- use `protected` members if the outside world doesn't have a need to know, but subclasses might;
- public access should only be allowed when it is absolutely necessary, and the access should be limited as much as possible. Generally, don't provide direct access to properties but rather provide access methods - provide, for example, only read access if a variable should never be written. This provides an extra level of protection and preserves flexibility for future changes.

8.15 Philosophy

Notice that there has been a subtle yet profound shift in the way we are working with variables. In a traditional programming language variables are either global, with too broad a scope, or local, with too narrow a scope. Global variables often get woven through the fabric of the program, with no way, in large programs especially, to understand the ramifications of making changes, or to keep programmers from relying on the implications and side effects of the specific implementation. Local variables have the opposite problem. They are so restricted in scope that you are always having to pass them explicitly, cluttering the code with redundant declarations and intermediate variables.

In the object-oriented paradigm, we have created two intermediate classes of variables - much more visible than the old local variables yet much less visible than global variables. Variables which are declared `local` in the object-oriented paradigm are generally available, but only within a very carefully circumscribed context - only to the methods of the variable's immediate object. If your classes are well drawn, this is exactly the code that you want to have access to this type of variable.

This scope can also be broadened a bit more, by declaring the variable `static`. Now the variable becomes available to every object in the class, but it is still not available to the majority of the code, the code that lies outside the class.

In object-oriented programming, the language has become much more careful with the way variables are handled, while at the same time becoming more terse, more expressive and easier to read and understand.

8.16 Abstract Classes and Virtual Methods

Often we create a set of classes that can be viewed as all derived from a common base class. For example, we might start with a common base class of type `BasePacket` that sets out the structure of packets but is incomplete; we would never want to instantiate it. From this base class, though, we might derive a number of useful subclasses: Ethernet packets, token ring

packets, GPSS packets, satellite packets. Each of these packets might look very similar, all needing the same set of methods, but they could vary significantly in terms of their internal details.

We start by creating the base class that sets out the prototype for these subclasses. Since we don't need to instantiate the base class, we declare it to be *abstract* by declaring the class to be *virtual*:

```
virtual class BasePacket {
```

By themselves, abstract classes are not tremendously interesting, but abstract classes can also have *virtual* methods. Virtual methods provide prototypes for subroutines, all of the information generally found on the first line of a method declaration: the encapsulation criteria, the type and number of arguments, and the return type if it is needed. Later, when subclasses override virtual methods, they must follow the prototype exactly. Thus, all versions of the virtual method will look identical in all subclasses:

```
virtual class BasePacket {
    virtual protected function integer send(bit[31:0] data);
}

class EtherPacket extends BasePacket {
    protected function integer send(bit[31:0] data) {
        // body of the function
        ...
    }
}
```

EtherPacket is now a class we can instantiate. In general, if an abstract class has several virtual methods, all of the methods must be overridden for the subclass to be instantiable. If all of the methods are not overridden, the subclass needs to be abstract.

Methods of normal classes can also be declared virtual. In this case, the method must have a body. If the method does have a body, then the class can be instantiated, as can its subclasses. However, if the subclass overrides the virtual method, then the new method must exactly match the superclass's prototype.

8.17 Finding the Right Method

There are several subtleties that arise when we start using virtual methods, although the underlying rules are quite simple. At some point, the compiler or the run-time system needs to find the proper method. If we have a simple case, with no inheritance, then the answer is obvious.

```
GigaEtherPacket p = new;
p.send();
```


We will invoke the `send()` method declared in `GigaEtherPacket`. But if we have inherited a virtual method, we need to find the right version. If `GigaEtherPacket` is a subclass, and doesn't declare `send()`, where do we go?

The first step is to decide where in the class hierarchy to begin searching for the method. The rule is simple: we begin searching from the class associated with the handle for the method we need to find:

- in the case above, this would be the class `GigaEtherPacket`.
- if the method is being invoked from inside another method, then the handle is the invoking method's class; if `send()` now invokes `setup()`, we would start with the class containing `send()`:

```
task send() {
    setup();
}
```

the handle for this reference to `setup()` is, of course, the implicit handle "this":

```
task send() {
    this.setup();
}
```

The second step is to search the hierarchy:

- if the method is not defined at this level, begin going up the inheritance tree, unless
 - the method is defined in the superclass as local, then the inheritance chain is broken (you can't inherit this method) and we have an error, or
 - we are at the base class, in which case the method was not found
- if the method is defined but is not virtual, use it.
- otherwise, it must be virtual. Search from this class down the inheritance tree.
 - if you find a non-virtual method, use it.
 - if you hit the bottom of the tree, use the most recent virtual method with a body; this will be the method closest to the bottom of the tree. If there is none, the search failed.

This search criteria is straightforward. Go up the inheritance tree until you find a method you can use. If it is virtual, go back down the tree until you find a non-virtual method. If you hit bottom without finding one, then use the last virtual method with a body you came across.

For example, if we have

```
class BasePacket {
    virtual task send (integer value);
    task init() { send(0); } // calls a virtual task
}
class EtherPacket extends BasePacket {
{
    // no re-declaration of send() or init()
    ...
}
```

```

    }
    class Ether100 extends EtherPacket {
    {
        task send(integer value) { ... }
    }
    Ether100 ep = new;

```

Now, if we invoke `ep.init()`, the system will execute the version of `init()` defined in `BasePacket`, but the version of `send()` declared in `Ether100`.

Note: If a method is declared virtual in a base class, it is usually a good idea to declare it virtual in the subclasses, too.

8.18 Polymorphism: Dynamic Method Lookup

Polymorphism allows us to use superclass variables to hold subclass objects, and to reference the methods of those subclasses directly from the superclass variable. As an example, consider the base class for our packet objects, `BasePacket`. Assume that it defines, as virtual functions, all of the public methods that are to be generally used by its subclasses, methods such as `send`, `receive`, `print`, etc. Even though `BasePacket` is abstract, we can still use it to declare a variable:

```
BasePacket packets[100];
```

We can now create instances of various packet objects, and we can put these into the array we just created:

```

EtherPacket ep = new;
TokenPacket tp = new;
GPSSPacket gp = new;
packets[0] = ep; packets[1] = tp; packets[2] = gp;

```

If our data types were, for example, integers, bits and strings, we couldn't store all of these types into a single array, but with *polymorphism* we can with objects. In this example, since the methods were declared as virtual, we can access the appropriate subclass methods from the superclass variable even though the compiler didn't know - at compile time - what was going to be loaded into, for example, `packets[1]`.

```
packets[1].send();
```

will invoke the `send` method associated with the `TokenPacket` class. At run-time, the system correctly binds the method from the appropriate class.

This is a typical example of polymorphism at work, providing capabilities that are far more powerful than what is found in a non-object-oriented language.

8.19 Out of Block Declarations

It is generally good coding practice to keep the class declaration to about a page. This makes the class easy to understand and to remember; declarations that go on for pages are hard to follow, and it is easy to miss short methods buried among the multi-page declarations.

To make this practical, it is best to move long method definitions out of the body of the class declaration. You do this in two steps. Within the class body, you declare the method prototype - whether it is a function or task, any attributes (local, protected, public, and/or virtual), and the full specification of its arguments. Then, outside of the class, you declare the full method - including the prototype but without the attributes - and, to tie the method back to its class, you qualify the method name with the class name and a pair of colons:

```
class Packet
{
    Packet next;
    function Packet get_next() { get_next = next; } // single line
    protected virtual function integer send (integer value);
}

function integer Packet::send(integer value)
{ // dropped protected virtual, added Packet::
    // body of method
    ...
}
```

The first lines of each part of the method declaration are nearly identical, except for the attributes and class-reference fields.

8.20 External Classes

As with subroutines, the class declaration can be in a separate file from the code that instantiates and invokes the class; you need to provide an external declaration of the class to support the tight type-checking required by Vera. The attributes and the method prototypes need to be re-declared:

```
extern class packet {
    bit [3:0] command;
    bit [40:0] address;
    bit [4:0] master_id;
    task issue_request( integer delay );
    function integer current_status();
}
```

Only the local and public members of the class must be listed in the extern class declaration.

Note – The `-h` flag will cause the Vera compiler to automatically generate the extern class declarations for you, in a `.vrh` file.

8.21 Typedef

Sometimes you need to declare a class variable before the class itself has been declared. For example, two classes may each need a handle to the other. When, in the course of processing the declaration for the first class, the compiler encounters the reference to the second class, that reference is undefined and the compiler flags it as an error. The way around this is to use `typedef` to provide an interim declaration for the second class:

```
typedef class C2;      // C2 is declared to be of type class
class C1 {
    C2 c;
}
class C2 {
    C1 c;
}
```

So, C2 is of type `class`, a fact that is re-enforced later in the source code.

Example 3: Object-Oriented Programming

This is an example of the arbiter in Example 2 using the object-oriented methodology. The example includes these features:

- Classes and objects
- Class properties
- Subclasses and Inheritance
- External classes

This example includes these files:

- *arb.if.vr*
- *main.vr*
- *classes.vr*
- *drive_tasks.vr*
- *RUN* (a run script)

The *RUN* script generates header files for *classes.vr* and *drive_tasks.vr*.

arb.if.vrh

```
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE

interface arb
{
    output [31:0] mem_addr PHOLD OUTPUT_SKEW;
    inout [15:0] mem_data INPUT_EDGE PRZ OUTPUT_SKEW vca rz; // vca monitor
    output oe PHOLD OUTPUT_SKEW;
    output ce PHOLD OUTPUT_SKEW;
    input ack INPUT_EDGE vca r0; // vca monitor
    input clk CLOCK;
}

port generic_port
{
    addr;
    data;
    oe;
    ce;
    ack;
}
```

```

bind generic_port port_a
{
    addr arb.mem_addr;
    data arb.mem_data;
    oe arb.oe;
    ce arb.ce;
    ack arb.ack;
}

bind generic_port port_b
{
    addr arb.mem_addr;
    data arb.mem_data;
    oe arb.oe;
    ce arb.ce;
    ack arb.ack;
}

```

main.vr

```

#include <vera_defines.vrh>
#include "arb.if.vrh"
#include "classes.vrh"

program classes_test
{
    integer sem_id;
    B_RD_Packet_Type rd_pack[];
    B_WR_Packet_Type wr_pack[];

    // allocate one semaphore for exclusive access to memory

    sem_id = alloc ( SEMAPHORE, 0, 1, 1 );
    fork
    {
        // port_a stuff
        integer i;

        // generate 10 wr-packets to write to 10 locations

        for ( i = 0; i < 10; i++ )
        {
            wr_pack[i] = new;
            do_whatever with port_a ( wr_pack[i] );
        }

        // generate 10 rd-packets to read from 10 locations
    }
}

```

```

        for ( i = 0; i < 10; i++ )
        {
            rd_pack[i] = new;
            rd_pack[i].addr = wr_pack[i].addr;
            do_whatever with port_a ( rd_pack[i] );
        }
    }
    {
// port_b stuff
    integer i;

// generate 10 wr-packets to write to 10 locations
    for ( i = 10; i < 20; i++ )
    {
        wr_pack[i] = new;
        do_whatever with port_b ( wr_pack[i] );
    }

// generate 10 rd-packets to read from 10 locations
    for ( i = 10; i < 20; i++ )
    {
        rd_pack[i] = new;
        rd_pack[i].addr = wr_pack[i].addr;
        do_whatever with port_b ( rd_pack[i] );
    }
    }
    join all
    printf ("\n\n----- All Finished ----- \n");
} //end program

task do_whatever with generic_port ( Action_Packet_Type apt )
{
// these virtual methods call mechanism will propagate down
// and find the right version for the derived obj

    apt.action with ( get_bind() ) ();
    apt.print_members();
} //end task do_whatever

```

classes.vr

```

#include <vera_defines.vrh>
#include "arb.if.vrh"
#include "drive_tasks.vrh"

```

```

extern integer randseed;

// Virtual base class with 2 virtual function
// for inherited classes to customize on their own.

virtual class Action_Packet_Type
{
    virtual task print_members();
    virtual task action with generic_port ();
}

// Derived class for action type of burst-read

class B_RD_Packet_Type extends Action_Packet_Type
{
    static integer inst_num = 0;
    integer inst_id;
    bit [15:0] data [4];
    bit [31:0] addr;

    task new ()
    {
        addr = random ( randseed ) & 8'hfc; // 4-bytes aligned
        inst_id = inst_num++;
        printf ("B_RD_Packet_Type inst %0d is created\n", inst_id );
    }

    task print_members ()
    {
        integer i;
        printf ("B_RD_Packet_Type inst %0d has the following members:\n",
            inst_id );
        printf ("      addr : %h\n", addr );
        for ( i = 0; i < 4; i++ )
            printf ("      data[%0d] : %h\n", i, data[i] );
        printf ("\n\n");
    }

    task action with generic_port ()
    {
        printf ("B_RD_Packet_Type inst %0d will do a burst read\n", inst_id);
        do_read with (get_bind()) ( addr, data );
    }
}

//end B_RD_Packet_Type defn

// Derived class for action type of burst-write

```



```

class B_WR_Packet_Type extends Action_Packet_Type
{
    static integer inst_num = 0;
    integer inst_id;
    bit [15:0] data [4];
    bit [31:0] addr;

    task new ()
    {
        integer i;
        addr = random ( randseed ) & 8'hfc; // 4-bytes aligned
        for ( i = 0; i < 4; i++ )
            data[i] = random ( randseed );
        inst_id = inst_num++;
        printf ("B_RD_Packet_Type inst %0d is created\n", inst_id );
    }

    task print_members ()
    {
        integer i;
        printf ("B_RD_Packet_Type inst %0d has the following members:\n",
            inst_id );
        printf ("      addr : %h\n", addr );
        for ( i = 0; i < 4; i++ )
            printf ("      data[%0d] : %h\n", i, data[i] );
        printf ("\n\n");
    }

    task action with generic_port ()
    {
        printf ("B_WR_Packet_Type inst %0d will do a burst write\n",
            inst_id);
        do_write with (get_bind()) ( addr, data );
    }
}

//end class B_WR_Packet_Type defn

```

drive_tasks.vr

```

#include <vera_defines.vrh>
#include "arb.if.vrh"

extern integer sem_id;

// do_read : acquire semaphore and do burst read at the addr
// release semaphore by the end

```

```

task do_read with generic_port ( bit [31:0] addr, var bit [15:0] data[4] )
{
    // acquiring lock on to driving/sampling signals

    semaphore_get ( WAIT, sem_id, 1 );
    @1 $addr <= addr; // drive the addr, oe, ce bits
    $oe = 1'b1;
    $ce = 1'b1;
    @2, 10 $ack == 1'b1; // ack should have come within 8 cycles
    fork
    {
        @0, 3 $ack == 1'b1; // ack should stay for 4 cycles
        @0 $oe <= 1'b0; // at the end, deassert oe and ce
        @0 $ce = 1'b0;
    }
    {
        integer i;
        for (i = 0; i < 4; i++) // latch the data for 4 consecutive cycles
        {
            @(posedge CLOCK);
            data[i] = $data;
        } //end for
    }
    join all
    @1 $oe = void; // void drive to kill one more cycle

    // releasing lock

    semaphore_put ( sem_id, 1 );
} //end do_read

// do_write : acquire semaphore and do burst write at the addr
// release the semaphore by the end

task do_write with generic_port ( bit [31:0] addr, bit [15:0] data[4] )
{
    // acquiring lock on to driving/sampling signals
    semaphore_get ( WAIT, sem_id, 1 );
    @0 $addr <= addr; // drive the addr, oe, ce bits
    $oe = 1'b0;
    $ce = 1'b1;
    @2, 10 $ack == 1'b1; // ack should have come within 8 cycles
    fork
    {
        @0, 3 $ack == 1'b1; // ack should stay for 4 cycles
        @0 $ce = 1'b0;
    }
}

```

```
{
    integer i;
    for (i = 0; i < 4; i++) // drive the data for 4 consecutive cycles
    {
        $data = data[i] async;
        @(posedge CLOCK);
    } //end for
}
join all
@1 $oe = void; // void drive to kill one more cycle

// releasing lock

    semaphore_put ( sem_id, 1 );
} //end do_write
```

RUN

```
#!/bin/csh -f
rm -f core *.vshell *.vro >! /dev/null
vera -cmp -g -h drive_tasks.vr
if ($status) exit 1
vera -cmp -g -h classes.vr
if ($status) exit 1
vera -cmp -g main.vr
if ($status) exit 1
```

176 Example 3: Object-Oriented Programming

9. Automated Stimulus Generation

This chapter details Vera's automated stimulus generation. It focuses on constraint-driven stimulus generation and data packing. This chapter contains these sections:

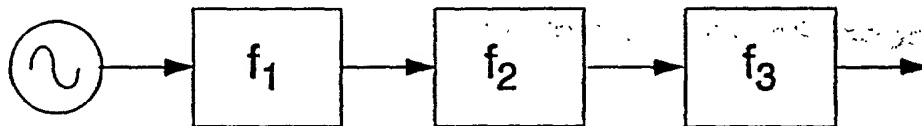
- Introduction to Stimulus Generation
- Stimulus Generation Overview
- Random Packet Generation
- Data Packing and Unpacking

9.1 Introduction to Stimulus Generation

Efforts to manually generate tests sufficient for verifying modern circuit designs have been rendered almost completely ineffective. The sheer complexity of designs, involving hundreds of valid states and transitions, makes deterministic tests inconceivable. The burden of generating stimuli to drive simulations must fall squarely on the verification tool.

The solution resides not in data sets that drive simulation testbenches but in random stimuli that ensure complete coverage of the hardware design. However, even this answer is not sufficient to efficiently and completely test a design. The ability to narrow the scope of the random stimuli such that test cases are not repeated and unnecessary tests are not performed can significantly speed up the verification process.

To meet this end, Vera makes use of constraint-driven random stimulus generation. Borrowing from classical signal-processing models, Vera uses a data source and a series of filters to transform and manipulate random data feeds into the form required to execute adequate tests.



The signal source is a random-number generator, while the filters can be configured to give any characteristic required by the data set. These facilities lead to test sets that are easy to understand and straightforward to develop.

Vera incorporates these stimulus generators into its object-oriented framework, creating a succinct, elegant, and powerful means of capturing data sets with an intrinsically random structure. Further, as objects, multiple stimulus generators can be instantiated and randomized independently.

Having extended objects to incorporate random variables, Vera also delivers a mechanism for packing and unpacking key members of an object. This provides a simple, straightforward mechanism for doing the type of data re-organization, such as generating, sending, receiving and analyzing data packets, that are common within many test environments.

9.2 Stimulus Generation Overview

Vera stimulus generation is divided into two major components: packet generation and data packing. Each is used within the object-oriented methodology to facilitate its use and re-use.

Vera's random packet generation uses the class constructs within object oriented programming. Class variables normally associated with standard packet generation are declared as random variables. Then each instance of the packet generator is randomized by calling Vera's predefined randomization method. This enables you to create multiple stimulus generators, randomized separately, using a single generation class.

Once Vera generates random stimuli, these stimuli are then filtered through constraint blocks. Constraint blocks are analogous to signal filters. They control the range of values that randomized variables can assume. Their use allows you to narrow the focus of tests to achieve adequate coverage with a maximum efficiency.

In addition to random packet generation, Vera provides a means to set random variables to all the possible boundary conditions. This provides another mechanism with which you can control and direct test generation.

The second component of Vera's automated stimulus generation capabilities is the packing and unpacking of data into bit streams. Vera again makes use of the object-oriented framework by including class methods to handle the packing and unpacking of data. Further, Vera defines several attributes for class members that are used to determine how data is packed and unpacked.

Together, these two components provide the means to generate and manage randomized data feeds in a controlled manner.

9.3 Random Packet Generation

Random packet generation is a specific implementation of Vera's packet generation capabilities. It consists of two major aspects: random variables and constraint blocks.

9.3.1 Random Variables

Variables are made random using the `rand` and `randc` keywords in the variable declaration. The syntax to declare a random variable is:

```
rand variable;  
randc variable;
```

Vera supports the randomization of variables of type integer, bit, and enumerated type. You can also randomize objects as well as arrays of integers, bits, enumerated types, and objects.

Associative arrays can be randomized, but you must specify the size of the array at runtime. For example:

```
rand type array_name[] assoc_size num_bytes;
```

num_bytes - *Num_bytes* specifies the number of elements in the array that will be randomized. Note that the array size itself can be a random number. You can use elements of the array other than those specified, but only the specified elements are randomized.

9.3.1.1 rand and randc

Vera has two forms of random variables, which are specified with the **rand** and **randc** keywords in the variable declaration.

Variables declared with the **rand** keyword are standard random variables. Their values will change randomly, and repeat values are allowed.

Variables declared with the **randc** keyword are random-cyclic variables. A random-cyclic variable takes on all possible values before repeating any value. Random-cyclic variables can only be bit or enumerated types, and are limited to a maximum size of 8 bits. This limits the range of values to those falling between 0 and 255.

9.3.1.2 Active and Inactive Random Variables

Vera provides the predefined **rand_mode()** method to control whether a random variable is active or inactive. All random variables are initially active. The syntax for the **rand_mode()** method is:

```
object_name.rand_mode(switch [, variable_name [, index] 1);
```

object name - The *object_name* is the name of the object in which the random variables are defined.

switch - The *switch* is either OFF or ON. OFF sets the specified variables to inactive so that it is not enforced on subsequent calls to the **randomize()** method. ON sets the specified variables to active so that they are randomized on subsequent calls to the **randomize()** method.

variable_name - The *variable_name* is the name of the variables to be made active or inactive. The variable name can be the name of any variable in the class hierarchy. If no variable name is specified, the switch is applied to *all* variables within the specified object.

index - The *index* is an optional array index. Omitting the index results in all the elements of the array being affected by the call. A -1 index value is treated the same as index omission.

The **rand_mode()** method returns the value of *switch* (either OFF or ON) if the change is successful. If the specified variable does not exist within the class hierarchy, the method returns a -1. If the specified variable exists but is not declared as **rand** or **randc**, the function returns a -1.

A special case for the *switch* argument is the **REPORT** keyword. Using **REPORT** will return the state of the specified constraint (either **OFF** or **ON**). The variable name and array index (if the variable is an array) must be specified for a **REPORT**.

For example:

```
class Packet {
    rand integer source_value, dest_value;
    constraint filter1 {
        source_value > 2 * m;
        dest_value <= 2 * m + 17;
    }
}

Packet packet_a = new;
integer ret = packet_a.rand_mode (OFF, "source_value");
// ... other code
ret = packet_a.rand_mode (ON, "source_value");
```

This example first makes the variable *source_value* inactive (**OFF**) and returns **OFF** to the variable *ret*. Then it makes the variable *source_value* active (**ON**) and returns **ON** to the variable *ret*.

9.3.2 randomize()

Variables are randomized using the **randomize()** class method. The syntax to randomize an object is:

```
object_name.randomize();
```

object_name - The *object_name* is the name of the object in which the random variables have been declared.

The **randomize()** class method generates random values for all random and random-cyclic variables within the specified class instance. If there are random objects within the class instance, calling the **randomize()** method also invokes their *object.randomize()* methods. Thus, all random variables and objects within a random object are randomized when the class instance is randomized.

When associative arrays are randomized, only elements within the specified range are randomized. Elements outside of this range are not randomized.

Random variables declared as **static** are shared by all instances of the class in which they are declared. Each time the **randomize()** method is called, the variable is changed in every class instance.

The **randomize()** method returns a 1 if it successfully sets all the random variables and objects to valid values. If it does not, it returns a 0, which typically indicates a problem with the constraints. If the **randomize()** call fails, any random variables set up until the failure retain their setting. If an object has no random variables anywhere in its inheritance hierarchy (no random variables or sub-objects) or if all of its random variables are inactive, the **randomize()** function returns a 1.

The `randomize()` method is implemented using the `random()` system function. Thus, executing `random(seed)` affects the sequence of random numbers generated by `randomize()`.

For example:

```
class Packet {
    local integer sequence;
    local randc bit [5:0] dest_addr;
    protected rand integer src_addr;
    static rand integer time_origin;
    rand Payload data_part;
}
```

This class definition declares a random-cyclic variable, `dest_addr`, a random variable, `src_addr`, a random variable, `time_origin`, and a random object, `data_part`. To randomize an instance of class `Packet`, call the `randomize()` method:

```
int success;
Packet p = new;
success=p.randomize();
```

9.3.2.1 Randomization Usage Details

The code for the pre-defined `randomize` method follows this pattern:

```
virtual function randomize (integer flag = 1)
{
    int success = 1;

    if (flag == 1)
        this.pre_randomize();

    if (flag == 1)
        // Randomize the rand and randc member variables (except
        // nested objects) of "this" object and all its ancestors,
        // subject to the active constraints.
        // Set success = 0 if it could not find a valid value for one.

    if (success == 1)
        success = super.randomize( 0 );

    while (success == 1)
        // For each nested object in "this" that is rand, do:
        success = nested_object.randomize( 1 );

    if (flag == 1)
        this.post_randomize();
```

```

        randomize = success;
    }
    task pre_randomize()
    {
        super.pre_randomize();
    }
    task post_randomize()
    {
        super.post_randomize();
    }

```

Notice that the `flag` parameter of `super.randomize` is 0, so that it will not call `pre_randomize()` and `post_randomize()`.

Also notice that the non-object member variables of "this" class instance and all its superclasses, back to the base class, are randomized at the same time. Random values are assigned in an order determined by the constraints, rather than by whether the `rand` or `randc` variable belongs to this class or a superclass. This allows enforcing a constraint like `"super.j < 2 * this.i;"`, where both `super.j` and `this.i` are random variables, and where `super.j` can be given its random value only after `this.i` has been given its random value.

Finally, notice that nested objects are randomized after the non-object variables. This allows variable-size arrays of nested objects, since the size of an associative array (as far as packing and randomizing are concerned) can depend on one or more random variables.

You can write your own version of the `randomize()` function. However, if you write your own version of `randomize()` for a particular class, the `rand` and `randc` variables in the class are not automatically set, and any constraints declared as part of the class are not enforced.

If your `randomize()` method calls the pre-defined `super.randomize()`, with the `flag` parameter set to 1 (or omitted), the `rand` and `randc` variables in this superclass and in its superclasses will be randomized, using the constraints in this superclass and its ancestors.

If the class with your `randomize()` has descendants with the pre-defined `randomize()`, calling the descendant's `randomize` function (with the `flag` parameter set to 1 or omitted) will affect variables in the descendant class and its superclasses, up to but not including the class with your `randomize()`, using the constraints in those classes. Your `randomize()` will be called, with the `flag` parameter set to 0.

The `randomize()`, `pre_randomize()` and `post_randomize()` methods must be public.

9.3.2.2 Using `randomize()` on Object Lists

Vera allows you to create lists of objects, which enables you to initialize non-random variables that constrain random variables.

This is an example of using the `randomize()` function on a list of objects:

```

class ListClass
{
    integer id;
    rand bit[3:0] b;
    rand ListClass next;

    task new (integer new_id)
    {
        id = new_id;
        next = null;
    }

    function ListClass append_one (integer new_id)
    {
        next = new (new_id);
        append_one = next;
    }
}

// These examples illustrate the order the list objects
// are randomized in.

task pre_randomize()
{
    printf("In ListClass pre_randomize: object %0d\n", id);
}

task post_randomize()
{
    printf("In ListClass post_randomize: object %0d: b is %0h\n", id, b);
}

task print_list()
{
    printf ("ListClass object %0d: b is %0h\n", id, b);
    if (next != null) next.print_list();
}

}

task DoList()
{
    integer i = 1;
    integer success;
    ListClass head, tail;

    // Create a list of objects with sequential id's.
    head = new( i++ );
    tail = head.append_one( i++ );
    tail = tail.append_one( i++ );
    tail = tail.append_one( i++ );
    tail = tail.append_one( i++ );
}

```

```

// Show the list, with variables not yet randomized.
printf("In DoList, the list before randomization is:\n");
head.print_list();
printf("\n");

// Invoke randomize on the head of the list. It will randomize
// any non-object member variables ("b" in this example), then it will
// invoke randomize on any object member variables ("next" in this
// example). This will be repeated automatically until it encounters
// the tail of the list, where "next" is null.

success = head.randomize();
if (success != OK) printf("randomize failed in DoList\n");

// Show the list, with variables set to random values.
printf("\nThe list after randomization is:\n");
head.print_list();
printf("\n");
}

```

9.3.2.3 Using randomize() to Create Arrays of Variable Size

Vera allows associative arrays, which can have very large indexes despite being a sparse array because elements are created when they are referenced. However, if an associative array is declared to be random, the number of its elements that are assigned values by the **randomize()** function must be specified. This expression can include random variables. Note that with this setup, constraints should be set up to force a non-negative value.

This is an example of using **randomize()** to create an array of variable size:

```

#define FACTOR 2
#define ORDINARY_SIZE 3
#define MAX_SMALL_ARRAY_SIZE 7

class SmallClass
{
    rand bit[2:0] small_var;
}

class ArrayClass
{
    // All these variables are random.
    rand
    {
        integer m, n;
        integer ordinary [ORDINARY_SIZE]; // ordinary array
    }
}

```

```

        bit [7:0] b_array [] assoc_size FACTOR * m; // associative array
        SmallClass small_array [] assoc_size n; // assoc. array of objects
    }

    // Constraint blocks are class members, just as the properties
    // (variables) and methods (tasks and functions) in a class are
    // class members. Constraints must follow the variables,
    // but come before the tasks and functions, of a class.

    constraint size_cons
    {
        m >= 0; m <= 3;
        n >= 1; n < MAX_SMALL_ARRAY_SIZE;
    }

    task new()
    {
        // Objects must be instantiated before they can be randomized.
        // Here, instantiate the maximum number of SmallClass objects
        // that might be randomized in small_array.

        integer i;

        for (i = 0; i < MAX_SMALL_ARRAY_SIZE; ++i)
            small_array[i] = new;
    }

    task print_array()
    {
        integer i;

        printf ("m is %0d, n is %0d\n", m, n);
        for (i = 0; i < ORDINARY_SIZE; ++i)
            printf ("ordinary[%0d] is %0d\n", i, ordinary[i]);
        for (i = 0; i < FACTOR * m; ++i)
            printf ("b_array[%0d] is %h\n", i, b_array[i]);
        for (i = 0; i < n; ++i)
            printf ("small_array[%0d].small_var is %b\n", i,
                small_array[i].small_var);
    }

    task DoArray()
    {
        integer success;
        ArrayClass a;

        a = new();
        success = a.randomize();
        if (success != OK) printf("randomize failed in DoArray\n");
    }

```

```

        printf ("In DoArray, ArrayClass values after randomize() call:\n");
        a.print_array();
        printf("\n");
    }

```

9.3.3 Constraint Blocks

The values of random variables can be controlled using constraint blocks. Constraint blocks limit the set of legal values that random variables can assume. These constraints are enforced by the `randomize()` class method.

Constraint blocks are class members. They are defined within the class to be randomized. They must be defined after all the variable declarations in the class and before all the task and function declarations in the class. The syntax to declare a constraint is:

```
constraint constraint_name { constraint_expressions }
```

constraint_name - The *constraint_name* is the name of the constraint block.

constraint_expression - The *constraint_expressions* are the conditional expressions that limits random values. It is a series of expressions that are enforced when the class is randomized. Constraint expressions are of the form:

```
random_variable operator expression;
```

random_variable - The *random_variable* parameter specifies the variable to which the constraint is applied.

operator - The valid operators for constraints are: `<`, `<=`, `==`, `>=`, `>`, `!=`, `===`, `!==`, `=?=, and !?=. You can also use in, lin, and dist for distribution sets.`

expression - The *expression* parameters can be any valid Vera expression with the following exceptions:

- You cannot call a task or function.
- Pre- and post-operators, like `j++`, are not allowed.
- If the random variable is cyclic, the expression cannot contain any random variables.
- Cyclic constraints are not allowed (for example, `A < B; B < 2 * A`).
- The specified random variable cannot appear on both sides of the expression.
- The specified random variable must be a member of this class or one of its antecedents.
- Expressions cannot include nested objects of this class, its superclasses or descendants - to preclude recursion.
- Expressions cannot include random member variables of nested objects.

If constraints in different classes in the inheritance hierarchy have the same name, the constraint in the lowest level descendant is used.

This is an example of a basic constraint block:

```

class Packet {
    rand integer source_value;
    constraint filter1 {
        source_value < n + 3;
        source_value > 2 * m;
    }
}

```

This example defines a class `Packet`, declares the random variable `source_value`, and defines the constraint block `filter1`. The constraint block `filter1` limits the range of values of `source_value` to those values falling between $(n+3)$ and $(2*m)$, where n and m are arbitrary global variables.

9.3.3.1 Distribution Sets

Distribution sets specify ranges of values and may assign weighted probabilities to them. The syntax to define a distribution set is:

```
variable command {ranges};
```

variable - The *variable* is the random variable, which is controlled by the constraint.

command - The *command* can be `in`, `lin`, or `dist`.

ranges - Each of the *ranges* is a range of values specified by Vera expressions. Ranges are defined by specifying a low and high value, separated by a colon (:). If only a single value is specified, the range exists only of that value. Multiple ranges can be declared in the same definition using commas to separate them. For example:

```
data_addr in {5:7, 10, x+3:y*2};
```

Expressions must follow the same rules random expressions follow regarding operators and variables.

Note - A random variable can have only one distribution set (`in`, `lin`, or `dist`) associated with it.

in and lin

The `in` command limits the values of the specified random variable to those within the defined range. For example:

```
data_addr in {1:4, 6, 8:10};
```

This example declares a constraint such that the only values that `data_addr` can assume are 1 to 4, 6, and 8 to 10.

An alternative to the `in` command is the `lin` command. The `lin` command specifies ranges of values that the random variable *cannot* assume. For example:

```
data_addr !in {1:4, 6, 8:10};
```

This example declares a constraint such that `data_addr` can assume all values *except* 1 to 4, 6, and 8 to 10.

dist

The **dist** command limits the values of the specified random variable to those within the defined range, and it assigns weighted probabilities to the values. The **dist** command uses two operators to assign weights: **:=** and **:/**.

The **:=** operator assigns the weight to each element in the list. The **:/** operator distributes evenly the weight across the values of the range. For example:

```
data_addr dist {5:7 := 10, 9 := 20};
data_addr dist {5:7 :/ 60, 9 :/ 40};
```

The first declaration assigns a weight of 10 each to 5, 6, and 7, and it assigns a weight of 20 to 9. The second declaration distributes a weight of 60 across the range 5:7, which assigns a weight of 20 each to 5, 6, and 7. It also assigns a weight of 40 to 9. The probability that any single value is assigned to the variable is the individual weight divided by the total weight of the distribution. For example, using the first declaration, the probability that `data_addr` is assigned a value of 5 is 10/50 or 20%.

9.3.3.2 Constraint Errors

If constraints are set such that they can never be satisfied, a hard error occurs and the simulation terminates. This occurs with conflicting constraints that do not contain random variables or distribution sets that have no valid entries.

If constraints that contain random variables cannot be satisfied, a soft error occurs. A 0 is returned and the simulation continues. A subsequent call to the `randomize()` method may change the constraint variable, and the check is made again.

In several cases, constraints that can never be satisfied are treated as soft errors. These include: **!=**, **!?**, **in**, **lin**, and **dist** constraints.

9.3.3.3 Active and Inactive Constraints

Vera provides the predefined `constraint_mode()` method to control whether a constraint is active or inactive. All constraints are initially active. The syntax for the `constraint_mode()` method is:

```
object_name.constraint_mode(switch [, constraint_name]);
```

object name - The *object_name* is the name of the object in which the constraint block is defined.

switch - The *switch* is either **OFF** or **ON**. **OFF** sets the specified constraint block to inactive so that it is not enforced on subsequent calls to the `randomize()` method. **ON** sets the specified constraint block to active so that it is enforced on subsequent calls to the `randomize()` method.

constraint_name - The *constraint_name* is the name of the constraint block to be made active or inactive. The constraint name can be the name of any constraint block in the class hierarchy. If no constraint name is specified, the switch is applied to *all* constraints within the specified object.

The *constraint_mode()* method returns the value of *switch* (either OFF or ON) if the change is successful. If the specified constraint block does not exist within the class hierarchy, the method returns a -1.

A special case for the *switch* argument is the **REPORT** keyword. Using **REPORT** will return the state of the specified constraint (either OFF or ON). The constraint name must be specified for a **REPORT**.

For example:

```
class Packet {
    rand integer source_value;
    constraint filter1 {
        source_value > 2 * m;
    }
}

Packet packet_a = new;
integer ret = packet_a.constraint_mode (OFF, filter1);
// ... other code
ret = packet_a.constraint_mode (ON, filter1);
```

This example first makes constraint filter1 inactive (OFF) and returns OFF to the variable *ret*. Then it makes constraint filter1 active (ON) and returns ON to the variable *ret*.

9.3.3.4 Void Constraints

Vera allows you to assign a void value to constraints, which is useful using conditionals. The syntax is:

```
random_variable operator (conditional) ? expression : void;
```

conditional - The *conditional* can be any valid constraint expression.

If the conditional evaluates to true, the constraint is set to the specified expression. If the conditional evaluates to false, the constraint is made void, and the random variable is not affected by this constraint. Note that it could be affected by other active constraints.

This is an example of several constraints used together:

```
r1 >= (r2==0) ? 1 : void;
r1 >= (r2==1) ? 2: void;
r1 >= (r2==2) ? 4: void;
r1 >= (r2==3) ? 8: void;
r1 >= (r2>3) ? 16: void;
```

This example sets *r1* greater than or equal to 1 if *r2* is equal to 0. If *r2* is equal to 1, *r1* is set so that it is greater than or equal to 2. The progression continues through all of the conditions.

9.3.3.5 Static Constraint Blocks

You can define a constraint block as being static by including the `static` keyword in the definition. The syntax to declare a static constraint block is:

```
static constraint constraint_name { constraint_expression }
```

If a constraint block is declared as static, calls to `constraint_mode()` affect all instances of the specified constraint in all objects. So if a static constraint is set to OFF, it is OFF for all instances.

9.3.3.6 Constraint Interdependencies

In Vera, a constraint has a single random variable on the left side of an operator. The expression on the right side of the operator may include other random variables. The `randomize()` function sorts the random variables before randomizing to ensure that a random variable on the right side of a constraint has been assigned a value before assigning values to variables that depend on those constraints.

This is an example of using constraint interdependencies:

```
#define NUM_R 9

class ConstraintClass
{
    rand integer r[NUM_R];
    constraint con1
    {
        // r[1] and r[2] will be assigned values before r[0]
        r[0] >= r[1] + 2 * r[2];
        r[1] > 0;
        r[1] < 10;

        // r[3], r[4], r[5], r[8] will be assigned values before r[2]
        r[2] in { r[3] : r[3] + 4, r[4] - 2 : r[5], r[8] };
        r[3] > 0;
        r[3] < 5;
        r[4] > -20;
        r[4] < -10;
        r[5] > 3;
        r[5] < 12;

        // r[7] and r[8] will be assigned values before r[6]
        r[6] == r[7] / r[8];
        r[8] > 1;
        r[8] < 5;

        // r[8] will be assigned a value before r[7]
```

```

        r[7] > r[8];
    }
    task print_vars()
    {
        integer i;
        for (i = 0; i < NUM_R; ++i)
            printf("r[%0d] is %d\n", i, r[i]);
    }
}
task DoConstraint()
{
    integer success;
    ConstraintClass c;

    c = new();
    success = c.randomize();
    if (success != OK) printf("randomize failed in DoConstraint\n");
    printf("In DoConstraint,ConstraintClass values after randomize()
        call:\n");
    c.print_vars();
    printf("\n");
}

```

9.3.3.7 Dynamic Constraint Modification

There are several ways to dynamically modify constraints on randomization:

- Within a constraint, the `?:` construct can be used like an if statement to indicate which of two values should appear on the right side of the constraint.
- A constraint block can be made active or inactive by using the `constraint_mode()` function. Initially, all constraint blocks are active. Inactive constraints are ignored by the `randomize()` function.
- Random variables can be made active or inactive using the `rand_mode()` function. Initially, all `rand` and `randc` variables are active. Inactive variables are ignored by the `randomize()` function.
- The weights in a `dist` constraint can be changed, affecting the probability that particular values in the set are chosen.

This is an example of dynamically modifying constraints:

```

class DynamicClass
{
    bit[5:0] counter;
    rand {
        bit[7:0] byte;
        bit[3:0] nibble;
    }
}

```

```

        bit[15:0] half_word;
        bit[31:0] word;
    }

    // If counter is > 10, randomize will give nibble a value that
    // satisfies "nibble == 4'b10xx". If not, randomize will give
    // nibble a value that satisfies "nibble == 4'b0xx1".

    constraint con_question
    {
        nibble == ( (counter > 10) ? 4'b10xx : 4'b0xx1 );
    }
    #ifdef VERA_4_0

    // If nibble is 0, randomize() will produce a value for byte that
    // satisfies "byte == 3", that is, it will set byte to 3.
    // If nibble is 1 or 2, it will set byte to 5.
    // If nibble is 3 or 4, it will set byte to (nibble | 4'b1000).
    // If nibble is between 5 and 8, it will set byte to 7.
    // If nibble is greater than 8, it will set byte to a random value
    // less than 12.

    constraint con_void
    {
        byte == ( (nibble == 0) ? 3 : void );
        byte == ( (nibble == 1 || nibble == 2) ? 5 : void );
        byte == ( (nibble == 3 || nibble == 4) ? nibble | 4'b1000 : void );
        byte == ( (nibble >= 5 && nibble <= 8) ? 7 : void );
        byte < ( (nibble > 8) ? 12 : void );
    }
    #endif

    // The next two constraint blocks conflict with each other -- that is,
    // there is no possible value for word that can satisfy both con_zero
    // and con_ones. The caller will make sure that only one of these two
    // is active at a time.

    constraint con_zero
    {
        word == { half_word, byte, nibble, 4'b00xx };
    }

    constraint con_ones
    {
        word == { half_word, byte, nibble, 4'b11xx };
    }

```

```

task new()
{
    counter = 0;
}

// The pre_randomize() task itself will modify counter, which will
// affect which constraints apply to the random variables.

task pre_randomize()
{
    ++counter;
}

task print_vars()
{
    printf("counter is %d\n", counter);
    printf("half_word is %h\n", half_word);
    printf("byte is %h\n", byte);
    printf("nibble is %h\n", nibble);
    printf("word is %h\n", word);
}

#define LOOP_MAX 20

task DoDynamic()
{
    integer i, j, success;
    DynamicClass d;
    d = new();

    // One of the two conflicting blocks must be made inactive.
    // After half of the loop iterations below are completed, this
    // one will be re-activated and the other one made inactive.
    // (Calling constraint_mode(OFF) will return OFF unless it is
    // unable to find a constraint block with the specified name,
    // e.g. due to a spelling error.)

    if (d.constraint_mode (OFF, "con_ones") != OFF)
        printf ("constraint_mode(OFF) failed on con_ones\n");
    printf ("In DoDynamic, turned con_ones constraint OFF.\n\n");
    for (i = 0; i < LOOP_MAX; ++i)
    {
        success = d.randomize();
        if(success != OK)printf("randomize failed in DoDynamic, i %0d\n",i);
        printf("DynamicClass values after randomize() call with
            i=%0d:\n",i);
        d.print_vars();
    }
}

```

```

printf ("\n");
if (i == LOOP_MAX / 2)
{
    if (d.constraint_mode (ON, "con_ones") != ON)
        printf ("constraint_mode(ON) failed on con_ones\n");
    if (d.constraint_mode (OFF, "con_zero") != OFF)
        printf ("constraint_mode(OFF) failed on con_zero\n");
    printf ("\nTurned con_ones constraint ON, con_zero OFF.\n\n");
}
}

#ifdef VERA_4_0

// Now inactivate all the random variables, leaving them at whatever
// values they now hold, except "nibble". Any constraints with inactive
// random variables on the left side are ignored.

if (d.rand_mode (OFF) != OFF)
    printf ("rand-mode(OFF) failed\n");
if (d.rand_mode (ON, "nibble") != ON)
    printf ("rand-mode(ON) failed on nibble\n");
printf ("\nTurned all random variables except nibble OFF.\n\n");
for (j = 0; j < 4; ++j)
{
    success = d.randomize();
    if(success != OK)printf("randomize failed in DoDynamic, j %0d\n",j);
    printf ("DynamicClass values after randomize() call with
            j=%0d:\n", j);
    d.print_vars();
    printf ("\n");
}
#endif
}

```

9.3.4 Boundary Conditions

Vera's boundary condition capabilities are used to generate values that match the upper and lower limits of a random variable's valid value range. This is particularly useful because many design flaws occur at the boundary conditions. Boundary conditions are generated when the system function `boundary()` is called. The syntax to call the `boundary()` function is:

```
object_name.boundary(which);
```

object_name - The *object_name* is the name of the object on which the `boundary()` call is being made.

which - The *which* argument must be either **FIRST**, which sets the effected random values to their first boundary condition, or **NEXT**, which cycles through each subsequent boundary condition.

Boundary conditions for random variables are determined by the constraints on a random variable. For instance, examine this constraint block:

```
class foo {
    rand integer a, b, c;
    constraint c1 {
        a>=10; a<=20;
        b>=0; b<=100;
        c in {1:15, 32:63};
    }
}
```

For this constraint block, *a* has the boundary conditions 10 and 20, *b* has the boundary conditions 0 and 100, and *c* has the boundary conditions 1, 15, 32, and 63. This results in 16 different boundary condition combinations that would be generated with the `boundary()` system function.

The `boundary()` method treats `rand` and `randc` variables the same way. Note that you can also use random variables within the constraints so that boundary conditions are dependent on random values. However, if the values of the right-hand side of a constraint change while the `boundary()` method is being invoked, the boundary conditions are determined by the values at the time of the function call. Subsequent changes to the conditions do not affect the current call, but they will affect future calls.

When the `boundary()` function is called, all random variables within the specified class instance are set to their boundary conditions. If the *which* argument is set to `FIRST`, the first set of boundary conditions are used. If the *which* argument is set to `NEXT`, the next set of boundary conditions in the sequence is used. The sequence of boundary conditions is influenced by a number of factors, including the order of variable declaration and the order they are named in the constraints.

If there are random objects within the class instance calling `boundary()`, it also invokes `object.boundary()`. Thus, nested objects and extended classes are affected by calls to `boundary()`. Random variables declared as `static` are shared by all instance of the class in which they are declared. Each time the `boundary()` method is called, the variable is changed in every class instance.

When the `boundary()` function is called, it returns a 1 (or OK) each time it successfully sets the random variables to their boundary conditions. If one or more variables cannot be set due to conflicting restraints, it returns a 0 (or FAIL). If the last set of conditions has been set, it returns a 2 (or OK_LAST). If an object has no random variables anywhere in its inheritance hierarchy (no random variables or sub-objects) or if all of its random variables are inactive, the `boundary()` function returns a `NO_VARS`.

The `boundary()` function makes calls to `pre_boundary()` and `post_boundary()` much in the same way that `randomize()` calls `pre_randomize()` and `post_randomize()`.

This is an example of the `boundary()` system function:

```
program b
{
    integer success;
    my_object obj = new;
```

```

    success = obj.boundary(FIRST);
    while (success==OK) success = obj.boundary(NEXT);
    if (success!=OK_LAST) printf("Boundary problem (%0d)\n", success);
}

```

This example defines the object `obj`. The random variables within `obj` are set to their initial boundary conditions with the first `boundary()` call. The while loop sets them to their other boundary conditions.

9.4 Data Packing and Unpacking

Data packing consists of the packing of data fields into a serial bit stream and the subsequent unpacking of the bit stream to re-form a data structure. Data packing is particularly useful for sending packets over serial communication streams and then converting back to a packet structure.

Again, data packing is integrated into the object-oriented framework of Vera. Vera defines several class methods that are used to pack and unpack data. Vera also provides a set of attributes for class members that designate how data is to be packed and unpacked.

There are two main aspects of data packing: property attributes and the pack and unpack class methods.

9.4.1 Property Attributes

Vera provides several property attributes that designate how data is to be packed and unpacked. These attributes include:

- `little_endian`
- `big_endian`
- `bit_normal`
- `bit_reverse`
- `packed`
- `unpacked`

The attributes `little_endian`, `big_endian`, `bit_normal`, and `bit_reverse` can only be assigned to variables also assigned the `packed` attribute. These attributes can only be assigned to class variables; they cannot be assigned to class methods or constraints.

When variable attributes are nested, the lowest level attribute overrides the higher level attributes. For example:

```

packed bit_reverse
{
    integer i;
    bit_normal string str;
    unpacked
    {
        integer k;
    }
}

```



```

    }
    integer n;
}

```

In this example, *i* and *n* are packed **bit_reverse**, *str* is packed **bit_normal**, and *k* is **unpacked**.

little_endian

Variables assigned the **little_endian** attribute are packed least-significant byte first. This is part of the default attribute settings.

big_endian

Variables assigned the **big_endian** attribute are packed most-significant byte first.

bit_normal

Variables assigned the **bit_normal** attribute are packed most-significant bit of each byte first. This is part of the default attribute settings.

bit_reverse

Variables assigned the **bit_reverse** attribute are packed least-significant bit of each byte first.

9.4.2 Packing Methods

The **pack** and **unpack** methods can be used to pack member variables of type integer, bit field, string, enumerated type, and objects into an associative array of type bit, and vice versa. Strings are packed as a sequence of bytes, with a zero-byte terminator. Uninitialized or unknown integers are packed as 32 bits of 'x'. Enumerated types are packed in up to 4 bytes. An associative array must have a size field specified before it is packed. Anything not within the size specification is not packed.

The syntax to pack data is:

```
object_name.pack(array, index, left, right);
```

array - The *array* parameter specifies the array into which data is to be packed. The array can be an associative array of any bit width.

index - The *index* parameter specifies the *array* index at which to start packing.

left/right - The *left* and *right* parameters specify the number of bits on the left and right to leave unchanged in *array[index]*. Normally, you initialize them to 0 before calling **pack()**.

The **pack()** method returns the number of bits packed. It also updates *offset*, *left*, and *right* so that they can be used as arguments to subsequent **pack()** calls when packing multiple objects into a single stream.

This is an example of the **pack()** method:

```

integer nbits;
integer offset = 0;
integer left = 0;
integer right = 0;
bit [7:0] stream[];

nbits = packet_head.pack(stream, offset, left, right);
nbits+ = packet_body.pack(stream, offset, left, right);

```

When the `pack()` method is called, it begins by packing the first member it encounters. If an object's handle is encountered, then the `pack()` method of that class is invoked. Packing an object handle that is null produces a warning, and nothing is packed for that nested object; that object handle should also be null when the containing object is unpacked. If a nested object's handle is not null when it is packed, the handle should also be non-null before it is unpacked; unpacking does not call `new` to create objects.

9.4.3 Unpacking Methods

The unpacking methods are analogous to the packing methods. The syntax to unpack data is:

```
object_name.unpack(array, index, left, right);
```

The parameters have the same definitions as with `pack()`. The *array* parameter should be set to the array into which data was packed. The *index*, *left*, and *right* are normally initialized to 0. They are then updated with each call to `unpack()`. This allows you to unpack multiple objects from a single array.

You should generally unpack multiple objects from one array in the same order in which they were packed. For example:

```

x.pack(...);
y.pack(...);
z.pack(...);
x.unpack(...);
y.unpack(...);
z.unpack(...);

```

9.4.4 Pack and Unpack Example

This is an example of the `pack()` and `unpack()` methods:

```

#include <vera_defines.vrh>

class Serial_Data_Type
{
    static integer inst_num = 0;
    packed
    {

```

```

        rand bit [19:0] bit_data;
        string comment;
    }

    task new()
    {
        integer status;

        status = this.randomize();
        if ( !status )
            error ("Randomize failed!\n");
        inst_num++;
        sprintf (comment, "This is serial packet instance number %0d having
            data %b ", inst_num, bit_data );
    } //end task new
}

program packed_test
{
    Serial_Data_Type sdata_arr[5];
    bit data_stream[]; // does not have to be byte stream
    integer i, g_offset, g_left, g_right;

    printf ("\n\nPacking data.....\n");
    g_offset = 0; g_left = 0; g_right = 0;
    for ( i = 0; i < 5; i++ )
    {
        sdata_arr[i] = new;
        printf (" %s \n", sdata_arr[i].comment );
        void = sdata_arr[i].pack ( data_stream, g_offset, g_left,
            g_right );
    } //end for
    printf ("\n\nUnpacking data in order ..... \n");
    g_offset = 0; g_left = 0; g_right = 0;
    for ( i = 0; i < 5; i++ )
    {
        void = sdata_arr[i].unpack ( data_stream, g_offset,
            g_left, g_right );
        printf (" %s \n", sdata_arr[i].comment );
    } //end for
}

```

9.4.5 Details of Pack and Unpack

The code for the pre-defined pack() method follows this pattern:

```

virtual function integer pack (var bit[N:0] array[],
    var integer offset, var integer left, var integer right,
    integer flag = 1)
{
    integer nbits = 0;

    if (flag == 1)
        this.pre_pack();

    nbits = super.pack (array, offset, left, right, 0);

    // code to pack the member variables of "this" that are marked packed
    // -- in order, including nested objects -- adding to nbits

    if (flag == 1)
        this.post_pack();

    pack = nbits;
}
task pre_pack()
{
    super.pre_pack();
}
task post_pack()
{
    super.post_pack();
}

```

Notice that the `pack` method calls the `pack()` of its superclass before packing member variables of the current class. Then, unless the superclass's `pack()` method has been overridden, it calls its own superclass's `pack()`. Thus, the packing operation ultimately begins with the base class and works its way down the hierarchy, so that data associated with the base class is packed before data associated with its descendants. Notice also that the *flag* parameter of `super.pack` is 0, so that it does not call `pre_pack()` and `post_pack()`.

The `unpack()`, `pre_pack()`, and `post_pack()` methods follow this same pattern. Each of these methods must be public.

If you override the pre-defined method, you must specify a particular bit width, which will be inherited by all the class's descendants.

Example 4: Stimulus Generation

This is an example using automated stimulus generators. The example includes these features:

- Random packet generation
- Constraints

This example includes these files:

- *globaldef.h*
- *memCtl.if.vrh*
- *stimgen.vr*
- RUN (a run script)

The RUN script generates a header file for *stimgen.vr*.

globaldef.h

```
#ifndef FILE_GLOBALDEF_H
#define FILE_GLOBALDEF_H

enum MEM_SEG = MEM_SEG0, MEM_SEG1, MEM_SEG2;
enum COVERAGE_VAL_OPT = GENERIC_COV, STATE_COV, TRANS_COV;

#define W_NOOP 1
#define W_FLUSH_CACHE 2
#define W_BURST_READ 8
#define W_SINGLE_READ 4
#define W_BURST_WRITE 7
#define W_SINGLE_WRITE 3

#endif
```

memCtl.if.vrh

```
#ifndef INC_MEMCTL_IF_VRH
#define INC_MEMCTL_IF_VRH
#define INPUT_EDGE PSAMPLE
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
```

```

interface memCtl
{
    input [3:0] mc_state INPUT_EDGE ;
    inout [2:0] cmd INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW ;
    inout cache_hit INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW ;
    inout cache_dirty INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW ;
    input clk CLOCK;
} // end of interface memCtl

port mport
{
    mc_state;
    cmd;
    cache_hit;
    cache_dirty;
} //end mport

bind mport mbind
{
    mc_state memCtl.mc_state;
    cmd memCtl.cmd;
    cache_hit memCtl.cache_hit;
    cache_dirty memCtl.cache_dirty;
} //end bind

#endif

```

stimngen.vr

```

#include <vera_defines.vrh>
#include "memCtl.if.vrh"
#include "globaldef.h"

// Random Obj t_Addr

// consists of random type members concerning the address values

// the 3 constraints have to be exclusively turned on, to select
// the value of address being generated, otherwise, it will
// give error of conflicting constraints

class t_Addr
{
    rand bit [15:0] tag;
    randc bit [7:0] cache_idx;
    constraint addr_seg0_con
    {

```

```

        tag >= 16'h0000 ;
        tag < 16'h0100 ;
    }
    constraint addr_seg1_con
    {
        tag >= 16'h0100 ;
        tag < 16'h0200 ;
    }
    constraint addr_seg2_con
    {
        tag >= 16'h0200 ;
        tag < 16'h0300 ;
    }
} //end class t_Addr defn

// Random Obj which contains t_Addr type of obj as member
// members include the t_Addr type obj, cache_hit, cache_dirty
// member methods include
//   gen_addr : do a randomize on the addr obj
//   incre_addr : increase the addr value
//   exec_cmd : execute the command type
//   test_boundary : test all boundary conditions with a
//                   certain segment with a certain command

class t_CmdPacket
{
    // Non-Rand Stuff
    bit [2:0]    command;

    // rand members

    rand
    {
        t_Addr mem_addr;
        bit stim_cache_hit;
        bit stim_cache_dirty;
    }

    local task drive_memCtl with mport ();
    task new();
    task gen_addr ( MEM_SEG which_mem_seg );
    task incre_addr ();
    task test_boundary with mport (MEM_SEG which_mem_seg, bit[2:0] do_cmd);
    task exec_cmd with mport( bit [2:0] do_cmd);
} //end class t_CmdPacket defn

```

```

task t_CmdPacket::new ()
{
    mem_addr = new;
}

task t_CmdPacket::gen_addr ( MEM_SEG which_mem_seg )
{
    integer status;
    case ( which_mem_seg )
    {
        MEM_SEG0 :
        {
            void = mem_addr.constraint_mode ( ON, "addr_seg0_con");
            void = mem_addr.constraint_mode ( OFF, "addr_seg1_con" );
            void = mem_addr.constraint_mode ( OFF, "addr_seg2_con" );
        }
        MEM_SEG1 :
        {
            void = mem_addr.constraint_mode ( OFF, "addr_seg0_con" );
            void = mem_addr.constraint_mode ( ON, "addr_seg1_con" );
            void = mem_addr.constraint_mode ( OFF, "addr_seg2_con" );
        }
        MEM_SEG2 :
        {
            void = mem_addr.constraint_mode ( OFF, "addr_seg0_con" );
            void = mem_addr.constraint_mode ( OFF, "addr_seg1_con" );
            void = mem_addr.constraint_mode ( ON, "addr_seg2_con" );
        }
    }
    //end case
    status = this.randomize();
    if ( status == 0 )
        error ("Randomize failed!\n");
}

task t_CmdPacket::test_boundary with mport (MEM_SEG which_mem_seg,
    bit [2:0] do_cmd)
{
    integer status;
    case ( which_mem_seg )
    {
        MEM_SEG0 :
        {
            void = mem_addr.constraint_mode ( ON, "addr_seg0_con");
            void = mem_addr.constraint_mode ( OFF, "addr_seg1_con" );
            void = mem_addr.constraint_mode ( OFF, "addr_seg2_con" );

```



```

    }
    MEM_SEG1 :
    {
        void = mem_addr.constraint_mode ( OFF, "addr_seg0_con" );
        void = mem_addr.constraint_mode ( ON, "addr_seg1_con" );
        void = mem_addr.constraint_mode ( OFF, "addr_seg2_con" );
    }
    MEM_SEG2 :
    {
        void = mem_addr.constraint_mode ( OFF, "addr_seg0_con" );
        void = mem_addr.constraint_mode ( OFF, "addr_seg1_con" );
        void = mem_addr.constraint_mode ( ON, "addr_seg2_con" );
    }
} //end case
status = this.boundary (FIRST);
if ( status != FAIL && status != OK_LAST )
    exec_cmd with (get_bind()) ( do_cmd );
else
    error ("unexpected failure in boundary call!\n");
while ( this.boundary ( NEXT ) != OK_LAST )
    exec_cmd with (get_bind()) ( do_cmd );
} //end test_boundary

task t_CmdPacket::incre_addr ()
{
    this.mem_addr.tag += 32;
} //end incre_addr

task t_CmdPacket::exec_cmd with mport ( bit [2:0] do_cmd )
{
    integer status;
    this.command = do_cmd;
    drive_memCtl with (get_bind()) ();
} //end task exec_cmd

task t_CmdPacket::drive_memCtl with mport ()
{
    @1 $cmd = this.command;
    @0 $cache_hit = this.stim_cache_hit;
    @0 $cache_dirty = this.stim_cache_dirty;
    @1 $mc_state == void;
} //end task drive_memCtl

```

RUN

```
#!/bin/csh -f
rm -f *.vro core *.vshell >& /dev/null
vera -cmp -g -h stimgen.vr
if ($status) exit 1
```

Copyright © 2000 Synopsys Inc.

10. Vera Stream Generator

This chapter introduces the Vera Stream Generator. It describes the powers of the VSG and details its use. This chapter includes these sections:

- VSG Overview
- Production Definitions
- Production Controls
- Value Passing

10.1 VSG Overview

The syntax for programming languages is often expressed in Backus Naur Form (BNF) or some derivative thereof. Parser generators use this BNF to define the language to be parsed. However, it is possible to reverse the process. Instead of using the BNF to check that existing code fits the correct syntax, the BNF can be used to assemble code fragments into syntactically correct code. The result is the generation of pseudo-random sequences of text, ranging from sequences of characters to syntactically and semantically correct assembly language programs.

Vera's implementation of a stream generator, the VSG, is defined by a set of rules and productions encapsulated in a `randseq` block. The general syntax to define a VSG code block is:

```
randseq (production_name) {  
    production_definition1;  
    production_definition2;  
    ...  
    production_definitionN;  
}
```

When the `randseq` block is executed, random production definitions are selected and streamed together to generate a random stream. How these definitions are generated is determined by the base elements included in the block.

Any VSG code block is comprised of production definitions. Vera also provides weights, production controls, and production system functions to enhance production usage. Each of these VSG components is discussed in detail in subsequent sections.

10.2 Production Definitions

A language is defined in BNF by a set of production definitions. The syntax to define a production is:

```
production_name : production_list;
```

production_name - The *production_name* is the reference name of the production definition.

production_list - The *production_list* is made up of one or more *production_items*.

10.2.1 Production Items

One or more production items makes up a production list. Production items are made of terminals and non-terminals.

A terminal is an indivisible code element. It needs no further definition beyond the code block associated with it. Code blocks should be encapsulated in braces ({ }). A non-terminal is an intermediate variable defined in terms of other terminals and non-terminals.

If a production item is defined using non-terminals, those non-terminals must then be defined in terms of other non-terminals and terminals using the production definition construct. Ultimately, every non-terminal has to be broken down into its base terminal elements.

Multiple production items specified in a production list can be separated by white space or by the OR operator (|). Production items separated by white space indicate that the items are streamed together in sequence. Production items separated by the | operator force a random choice, which is made every time the production is called.

This is a simple example illustrating the use of production items:

```
main : top middle bottom;
top : one | two;
middle : three | four;
bottom : five;
```

The *main* production definition is defined in terms of three non-terminals: *top*, *middle*, and *bottom*. When the call is made to this random sequence, *top*, *middle*, and *bottom* are evaluated, and their definitions are streamed together.

The *top*, *middle*, and *bottom* production definitions are defined in terms of terminals. The | operator forces a choice to be made between the *one* and *two* terminals and also between the *three* and *four* terminals.

This sequence block leads to these possible outcomes:

```
one three five
one four five
two three five
two four five
```

This is an example of a full set of production definitions:

```

assembly_program: text_section data_section ;
text_section: {printf(".text");} my_code ;
data_section: {printf(".data");} data ;
data: initialized_data | uninitialized_data ;
my_code: { /* Vera code */}
initialized_data: { /* Vera code */ }
uninitialized_data: { /* Vera code */}

```

This example defines the production *assembly_program* in terms of *text_section* and *data_section*. The production *text_section* is then broken down to include the string ".text" and the non-terminal *my_code*. The production *data_section* is defined in terms of either of the two terminals *initialized_data* and *uninitialized_data*, the selection of which occurs when the randseq block is called. The resulting output is:

```

.text
my_code output
.data
initialized_data output OR uninitialized_data output

```

The *my_code*, *initialized_data*, and *uninitialized_data* outputs are determined by the code blocks associated with those productions.

10.2.2 Weights

Weights can be assigned to production items to change the probability that they are selected when the randseq block is called. The syntax to declare a weight is:

```
production_name : weight production_item;
```

weight - The *weight* must be in the form of *&(expression)* where the *expression* can be any valid Vera expression that returns a non-negative integer. Function calls can be made within the expression, but the expression must return a numeric value, or else a simulation error is generated.

Assigning weights to a production item affects the probability that it is selected when the randseq block is called. Weight should only be assigned when a selection is forced with the *!* operator. The weight for each production item is evaluated when its production definition is executed. This allows you to change the weights dynamically throughout a sequence of calls to the same production.

This is an example of a weighted production definition:

```

integer_instruction : &(3) add_instruction
                    | &(i*2) sub_instruction ;

```

This example defines the production *integer_instruction* in terms of the weighted production items *add_instruction* and *sub_instruction*. If *i* is 1 when the definition is executed, there is a 60% (3/5) chance that *add_instruction* is selected, and a 40% (2/5) chance that *sub_instruction* is selected.

10.3 Production Controls

Vera provides several mechanisms that can be used to control productions: **if-else** statements, **case** statements, and **repeat** loops.

10.3.1 If-else Statements

A production can be conditionally referenced using an **if-else** statement. The syntax to declare an **if-else** statement within a production definition is:

```
production_name : <if (condition) production_name else production_name>;
```

condition - The *condition* can be any valid Vera expression.

If the conditional evaluates to true, the first production item is selected. If it evaluates to false, the second production item is selected. The **else** statement can be omitted. If it is omitted, a false evaluation ignores the entire **if** statement.

This is an example of a production definition that uses an **if-else** statement:

```
assembly_block : <if (nestingLevel > 10) seq_block else any_block>;
```

This example defines the production *assembly_block*. If the variable *nestingLevel* is greater than 10, the production item *seq_block* is selected. If *nestingLevel* is less than or equal to 10, *any_block* is selected.

10.3.2 Case Statements

A general selection mechanism is provided by the **case** statement. The syntax to declare a **case** statement within a production definition is:

```
production_name : <case(primary_expression)
    case1_expression : production_name
    case2_expression : production_name
    ...
    caseN_expression : production_name
    default : production_name >;
```

expression - The *expressions* can be any valid Vera expression or comma-separated list of expressions.

The *primary_expression* is evaluated when the production definition is executed. The value of the *primary_expression* is successively checked against each *case_expression*. When an exact match is found, the production item corresponding to the matching case is executed, and control is then passed to the first line of code after the case block. If other matches exist, they are not executed. If no match is found, the **default statement** is executed. If no default statement is specified and no matches are found, control passes to the first line of code after the case statement without any production being executed.

This is an example of a production definition using a **case** statement:

```
assembly_block : <case(i*3)
    0 : seq_block
    3 : loop_block
    default : any_block> ;
```

This example defines the production *assembly_block* with a case statement. The primary expression *i*3* is evaluated, and a check is made against the case expressions. The corresponding production item is executed.

10.3.3 Repeat Loops

The repeat loop is used to loop over a production a specified number of times. The syntax to declare a repeat loop within a production definition is:

```
production_name : <repeat (expression) production_name>;
```

expression - The *expression* can be any valid Vera expression that evaluates to a non-negative integer, including functions that return a numeric value.

The expression is evaluated when the production definition is executed. The value specifies how many times the corresponding production item is executed.

This is an example of a production definition using a repeat loop:

```
seq_block : <repeat (random() ) integer_instruction>;
```

This example defines the production *seq_block*, which repeats the production item *integer_instruction* a random number of times, depending on the value returned by the *random()* system function.

10.3.4 Break and Continue

Vera provides the *break* and *continue* statements for use inside randseq blocks.

10.3.4.1 Break

The *break* statement is used to terminate a randseq block. The syntax to declare a *break* statement is:

```
break;
```

A *break* statement can be executed from a code block within a production definition. When a *break* statement is executed, the randseq block terminates immediately and control is passed to the first line of code after the randseq block.

This is an example of a production definition using a *break* statement:

```
SETUP_COUNTER: {
    integer regis = regFile.getRegister();
    integer value = ADDI;
    nestingLevel++;
```

```

        if (nestingLevel == MAX_NESTING) break;
        ...
    } ;

```

This example executes the break if the conditional is satisfied. When the break is executed, control passes to the first line of code after the randseq block.

10.3.4.2 Continue

The **continue** statement is used to interrupt the execution of the current production. The execution continues on the next item in the production from which the call is made. The syntax to declare a **continue** statement is:

```
continue;
```

The **continue** statement passes control to the next production item in the production from which the call is made without executing any code in between.

This is an example of a production definition using a **continue** statement:

```

LOOP_BLOCK: SETUP_COUNTER GEN_LABEL;

SETUP_COUNTER: {
    integer regis = regFile.getRegister();
    integer value = ADDI;
    nestingLevel++;
    if (nestingLevel == MAX_NESTING) continue;
    ...
} ;

```

This example first executes the *SETUP_COUNTER* production definition. When the **continue** is executed, the code after the **continue** is ignored and control passes to the *GEN_LABEL* production definition.

10.4 Value Passing

Value passing within randseq blocks allows you to associate a data type with each production definition in order to pass values between production definitions. There are two components of value passing within randseq blocks: value declaration and value passing functions.

10.4.1 Value Declaration

To associate a data type and value with a given non-terminal production, you must declare the production using the **prod** declaration. The syntax to declare a production for value passing is:

```
prod data_type production_name;
```

data_type - The *data_type* integer, bit, string, enumerated type, or any object.

production_name - The *production_name* is the name of the production that is passing the value.

Multiple productions can be declared in a single declaration statement, similar to variables. Vera performs strict type checking for passing values.

10.4.2 Value Passing Functions

Vera provides two functions used to pass values in randseq blocks: **prodset()** and **prodget()**. The **prodset()** and **prodget()** system functions can be called from Vera code blocks within production definitions.

prodset()

The **prodset()** system task is used to set a value associated with a non-terminal. The syntax to set a value is:

```
prodset (value [, production_name [, occurrence_number] ] );
```

value - The *value* is the value you want to pass. It must be of the same type as the **prod** declaration. It can be an integer, bit, string, enumerated type, or object.

production_name - The *production_name* optionally specifies the name of the non-terminal production the value is being assigned to. If it is omitted, the production that the task is called in is assumed to be the production to the left of the colon (:).

occurrence_number - The *occurrence_number* optionally specifies which occurrence of the same production name receives the value. If the same production is referred to multiple times in the same definition, the first is 1, and the others are numbered sequentially. If it is omitted, it is assumed to be 1.

The **prodset()** system task assigns the specified value to the specified occurrence of the non-terminal production. This value can be retrieved using the **prodget()** system function.

prodget()

The **prodget()** system function is used to retrieve values assigned to non-terminal productions. The syntax to pass a value is:

```
prodget ([production_name [, occurrence_number] ] );
```

production_name - The *production_name* specifies the name of the non-terminal production the value has been assigned to. If it is omitted, the production that the task is called in is assumed to be the production to the left of the colon (:).

occurrence_number - The *occurrence_number* optionally specifies which occurrence of the same production name receives the value. If the same production is referred to multiple times in the same definition, the first is 1, and the others are numbered sequentially. If it is omitted, it is assumed to be 1.

The **prodget()** system function returns the value assigned to the specified non-terminal production.

Production values can only be associated with production names to the left of the code block where `prodset()` is called. If you assign a value to the production name to the left of the colon (:), the value is passed up to the calling production, where it can be retrieved.

Calling `prodget()` on a production entry that has not been assigned returns an undefined value.

You cannot set a value for a production that has not yet been executed. For example:

```
main : prod_a {prodset(5,prod_b,1); } prod_b      /* invalid! */
main : prod_a {prodset(prod_a,2); } prod_b prod_a /* invalid! */
main : prod_a {integer tmp = prodget(prod_a);
               prodset(tmp+5,prod_a);} prod_b    /* valid! */
```

This is an example of how to pass values within a `randseq` block:

```
program GenList
{
    list vsgList;
    randseq()
    {
        prod list_node ELEMENT;

        TOP : { vsgList = new; } LIST END
        ;
        LIST : &(10) LIST ELEMENT
        {
            list_node lnode = prodget ( ELEMENT, 1 );
            vsgList.insert ( lnode );
            printf ("last insert %0d\n", lnode.data);
        }
        | ELEMENT
        {
            list_node lnode = prodget ( ELEMENT, 1 );
            vsgList.insert ( lnode );
            printf ("inserting %0d \n", lnode.data);
        }
        ;
        ELEMENT : {
            list_node lnode = new;
            prodset ( lnode );
            printf ("new node is created %0d\n", lnode.data);
        }
        ;
        END : { vsgList.printAll(); }
        ;
    }
}
```

This example defines the list object *vsgList*. When the *randseq* block is entered, the TOP production is executed. First, a list object is instantiated. Then the LIST production is executed. The LIST production consists of a weighted LIST ELEMENT production and an ELEMENT production. If the LIST ELEMENT production is selected, the LIST production is called recursively and the ELEMENT production is postponed. The original selection between ELEMENT and LIST ELEMENT is then made again. The cycle continues until the ELEMENT production is selected. At that time, *lnode* is assigned a value via the *prodset* call in the ELEMENT production. That value is inserted into *vsgList* via the *insert* call in the code block. Finally, the previously unexecuted ELEMENT calls that had been postponed when LIST ELEMENT was selected are executed. Control is then passed back to the TOP production, which executes the END production.

Note that each production is not assigned a single value. Instead, a stack of values can be assigned to a production, which are retrieved, in order, through the dynamic execution of the production set.

Copyright 2000 Synopsys, Inc. All rights reserved. Synopsys, the Synopsys logo, and Vera are registered trademarks of Synopsys, Inc. in the United States and other countries.

Example 5: Vera Stream Generator

This is an example using the Vera stream generator. The example includes these features:

- Random sequence blocks
- Productions
- Value passing

This example includes these files:

- *globaldef.h*
- *memCtl.if.vrh*
- *vsg.vr*
- *RUN* (a run script)

The *RUN* script generates a header file for *vsg.vr*.

Files included in the distribution but not shown here include:

- *coverage.vrh*
- *stimgen.vrg*

globaldef.h

```
#ifndef FILE_GLOBALDEF_H
#define FILE_GLOBALDEF_H

enum MEM_SEG = MEM_SEG0, MEM_SEG1, MEM_SEG2;
enum COVERAGE_VAL_OPT = GENERIC_COV, STATE_COV, TRANS_COV;

#define W_NOOP 1
#define W_FLUSH_CACHE 2
#define W_BURST_READ 8
#define W_SINGLE_READ 4
#define W_BURST_WRITE 7
#define W_SINGLE_WRITE 3

#endif
```

memCtl.if.vrh

```

#ifndef INC_MEMCTL_IF_VRH
#define INC_MEMCTL_IF_VRH
#define INPUT_EDGE PSAMPLE
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1

interface memCtl
{
    input [3:0] mc_state INPUT_EDGE ;
    inout [2:0] cmd INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW ;
    inout cache_hit INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW ;
    inout cache_dirty INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW ;
    input clk CLOCK ;
} // end of interface memCtl

port mport
{
    mc_state;
    cmd;
    cache_hit;
    cache_dirty;
} //end mport

bind mport mbind
{
    mc_state memCtl.mc_state;
    cmd memCtl.cmd;
    cache_hit memCtl.cache_hit;
    cache_dirty memCtl.cache_dirty;
} //end bind

#endif

```

vsg.vr

```

#include <vera_defines.vrh>
#include "RTL/memCtl.vh"
#include "memCtl.if.vrh"
#include "globaldef.h"
#include "stimgen.vrh"
#include "coverage.vrh"

```

```

extern t_MemCtl_Cov MemCtl_Cov; //coverage obj
extern t_CmdPacket CmdPacket; //command packet object
extern event evt_new_cmd; //global event variable, signalling new
                           //command is generated

// task generate_test:

// uses randseq{} as a sequence generator to create
// randomized sequence of commands out of the 6 command
// NOOP, FLUSH_CACHE, BURST_READ, BURST_WRITE, SINGLE_READ,
// SINGLE_WRITE. Environment conditions such as memory addresses
// and cache-hit status are generated during the sequence also.
// The productions are designed such that it will keep
// generating commands until coverage-goal is met by
// doing a query (GOAL) after each command

task generate_test with mport ()
{
    integer f_cnt;
    randseq()
    {
        prod integer FLUSH_A_LINE;
        STREAM : TRANSACTION
            <if ( !MemCtl_Cov.query (GOAL )) STREAM >
        TRANSACTION : GEN_ADDR SEL_CMD
        {
            trigger ( ONE_BLAST, evt_new_cmd );
        }
        ;
        GEN_ADDR :
        {
            randcase
            {
                1 : CmdPacket.gen_addr ( MEM_SEG0 );
                1.: CmdPacket.gen_addr ( MEM_SEG1 );
                1 : CmdPacket.gen_addr ( MEM_SEG2 );
            }
        }
        ;
        SEL_CMD : & ( W_NOOP ) DO_NOOP
            | & ( W_FLUSH_CACHE ) {f_cnt = 0; } DO_FLUSH_CACHE
            | & ( W_BURST_READ ) DO_BURST_READ
            | & ( W_SINGLE_READ ) DO_SINGLE_READ
            | & ( W_BURST_WRITE ) DO_BURST_WRITE
            | & ( W_SINGLE_WRITE ) DO_SINGLE_WRITE
        ;
    }
}

```

```

DO_NOOP :
{
    CmdPacket.exec_cmd with (get_bind())
    ( NO_OP );
}
;
DO_FLUSH_CACHE : FLUSH_A_LINE
    < if ( prodget ( FLUSH_A_LINE ) < 31 ) DO_FLUSH_CACHE >
;
FLUSH_A_LINE :
{
    CmdPacket.exec_cmd with ( get_bind() )
    ( FLUSH_CACHE );
    CmdPacket.incre_addr();
    prodset ( f_cnt++ );
}
;
DO_BURST_READ :
{
    integer wait_time;
    CmdPacket.exec_cmd with ( get_bind() )
    ( BURST_READ );
    if ( CmdPacket.stim_cache_hit )
        wait_time = 4 ;
    else if ( CmdPacket.stim_cache_dirty )
        wait_time = 6 ;
    else
        wait_time = 5;
    repeat ( wait_time ) @(posedge CLOCK);
}
;
DO_SINGLE_READ :
{
    integer wait_time;
    CmdPacket.exec_cmd with ( get_bind() )
    ( SINGLE_READ );
    if ( CmdPacket.stim_cache_hit )
        wait_time = 1 ;
    else if ( CmdPacket.stim_cache_dirty )
        wait_time = 3 ;
    else
        wait_time = 2;
    repeat ( wait_time ) @(posedge CLOCK);
}

```



```

;
DO_BURST_WRITE :
{
    integer wait_time;
    CmdPacket.exec_cmd with ( get_bind() )
    ( BURST_WRITE );
    if ( CmdPacket.stim_cache_hit )
        wait_time = 4 ;
    else if ( CmdPacket.stim_cache_dirty )
        wait_time = 6 ;
    else
        wait_time = 5;
    repeat ( wait_time ) @(posedge CLOCK);
}
;
DO_SINGLE_WRITE :
{
    integer wait_time;
    CmdPacket.exec_cmd with ( get_bind() )
    ( SINGLE_WRITE );
    if ( CmdPacket.stim_cache_hit )
        wait_time = 1 ;
    else if ( CmdPacket.stim_cache_dirty )
        wait_time = 3 ;
    else
        wait_time = 2;
    repeat ( wait_time ) @(posedge CLOCK);
}
;
} //end randseq
} //end task

```

RUN

```

#!/bin/csh -f
rm -f *.vro core *.vshell >& /dev/null
vera -cmp -g -h vsg.vr
if ($status) exit 1

```

Example 5: Vera Stream Generator

11. Functional Coverage

This chapter introduces Vera's functional coverage analysis and discusses how it is used. It includes these sections:

- Coverage Overview
- Coverage Definitions
- Instantiation and Triggering
- Coverage Feedback: The Query Function
- Coverage Administration
- Cross Coverage
- Backward Compatibility

11.1 Coverage Overview

As chip designs grow more complex and testing environments become increasingly sophisticated, emphasis is placed not only on testing the chip but testing it completely. With hundreds of possible states in a system and thousands of possible transitions, the completeness of tests needs to be a primary focus of any verification tool.

Traditional coverage models use a code coverage methodology. They check that specific lines of code are executed at some point in the simulation. This method has inherent flaws. For instance, you can be certain that a device entered states 1, 2 and 3, but you cannot be certain that the device transitioned from state 1 to 2 to 3 in sequence. Even such a simple example displays the limitations of code coverage. With a sophisticated chip, such coverage is not adequate to ensure the integrity of the design.

Vera supports a functional coverage system. This system is able to monitor all states and state transitions, as well as changes to variables. By setting up a number of monitor bins that correspond to states, transitions, and variable changes, Vera is able to track the activity in the simulation.

Each time a user-specified activity occurs, a counter associated with the bin is incremented. If you establish a bin for each state, state transition, and variable change that you want to monitor, you can check the bin counter after the simulation to see how many activities occurred. Thus, it is simple to check the degree of completeness of the testbench and simulation.

Vera further expands this basic functionality to include two analysis mechanisms: open-loop analysis and closed-loop analysis. Open-loop analysis monitors the bins during the simulation and writes a report at the end summarizing the results. Closed-loop analysis monitors the bins during the simulation and checks for areas of sparse coverage. This information is then used to drive subsequent test generation to ensure satisfactory coverage levels.

11.2 Coverage Definitions

To check the coverage for a given simulation, you create coverage definitions. For each coverage definition, you must define the monitor bins, initialization routines, and any triggers or other synchronization mechanisms you want to use. Meanwhile, Vera assigns pre-defined properties and methods to the coverage objects that take care of the data collection and much of the analysis. In this way, Vera successfully encapsulates a large portion of the code into reusable objects, which greatly facilitates their use.

Coverage definitions behave very similarly to classes that exist within the object-oriented methodology. Like classes, coverage *instances* have definitions. However, unlike a class definition, coverage definitions cannot contain methods and variables. Instead, they contain *declarations* of monitor bins.

Because coverage definitions behave very similar to classes within the object oriented methodology, they can be used in a similar manner. For instance, if you have four copies of the same bus protocol, you can instantiate four instances of the same coverage definition without creating four separate coverage definitions. When you call the coverage object within a Vera task, it inherits the virtual port to which the task is bound. So the same coverage definition can be used to check multiple ports or signals.

The basic syntax for defining a coverage definition is:

```
coverage_def Object_name (state_variable, arguments) {
    coverage declarations;
    measure of coverage;
    coverage goal;
    coverage option;
    initialization;
}
```

state_variable - The *state_variable* is the type and variable name being monitored by the coverage definition. It can be of the following types: integer, bit field (in the form bit[n1:0]), enumerated type, and port.

The *state variable* can be used as a parameter throughout the coverage definition.

arguments - The *arguments* are initialization parameters passed at instantiation. They have the same conventions as subroutine arguments and can have their default values set within the declaration. *Arguments* must be of the following types: integer, bit field, enumerated type and port variable.

The details of the remainder of the coverage definition are explained in subsequent sections.

Note - The coverage definition is just a definition. It does not instantiate a coverage object. To instantiate a coverage object, see Section 11.3, "Instantiation and Triggering."

11.2.1 Expressions within Coverage Definitions

Expressions within coverage definitions are used for specifying coverage declarations, measure of coverages, and coverage goals. These expressions are a subset of the general Vera expressions. Expressions must be one of the following data types: integer, bit, enum, port_var or interface signal. They cannot include arrays, the =, ++, or - operator, or function calls other than query(), get_cycle(), and get_time(). Function calls are only allowed in if expressions and coverage_value expressions. Expressions within coverage objects can reference global variables and arguments passed into the coverage definition.

11.2.2 Coverage Declarations

Coverage declarations are used to declare legal states, illegal states, legal transitions, and illegal transitions. They associate bins with these activities and monitor how many times these activities occur within a simulation.

11.2.2.1 State Declarations

The syntax for a state declaration is:

```
state state_bin_name (state_specification) conditional;
```

State Specification

In a state declaration, a single state or multiple states are associated with a monitor bin via a state specification. The state specification is a list of elements (separated by commas) that are matched against the current value of the state variable. For the current cycle, any matches increment the bin counter by one.

Each element of the state specification must be in one of the following formats:

1. *expression* - A counter is added to the bin when the state variable matches the expression exactly. 'x' or 'z' must match exactly.

2. *low:high* - A counter is added to the bin when the state variable matches any value in the range from *low* to *high*.
3. *low:high:step:repeat* - This creates multiple ranges. The first block ranges from *low* to *high*. The second block ranges from (*high+step*) to (*2*high-low+step*). New blocks are generated *repeat* times. For example: 2:5:10:3 would produce states {2, 3, 4, 5, 15, 16, 17, 18, 28, 29, 30, 31} (2,3,4,5) -> (15,16,17,18) -> (28,29,30,31)

Complex state specifications can be generated by separating multiple formats with commas. For example:

```
state state_bin_name (8'b0000_01XX, 8:10, 15:17:7:2)
```

This state specification will increment the bin counter if any of the specifications matches the state variable. In this example, a counter will be added to the bin if:

- (i) the state variable matches 8'b0000_01XX exactly
- (ii) the state variable falls in the range of 8 to 10
- (iii) the state variable falls in the range of 15 to 17 or 24 to 27

Note - 'x' and 'z' are not allowed in repeated range statements.

A special case of a state declaration uses the **all** state specification.

```
state state_bin_name (all);
```

This statement indicates that all values of the state variable increment the specified bin counter.

Note - If you use state(all), a significant amount of memory is used if the number of sampled and unique states is large.

It is important to note that state specifications are evaluated only once, at the time the coverage object is instantiated. While the state variable is being monitored, the state specification remains constant.

State Bin Names

Bins can be assigned explicit names, or the Vera compiler can generate implicit names. The Vera compiler generates implicit bin names based on the state specification. The general format is always prefaced by **s_**. The following characters are converted to underscores (**_**): comma (**,**), colon (**:**), slash (**/**), dash (**-**), plus (**+**), asterisk (*****), and period (**.**). The following characters are ignored when the bin name is generated: double quote (**"**), brackets (**[]**), parentheses (**()**), dollar sign (**\$**), caret (**^**), backslash (****), and braces (**{}**). For example:

state expression	implicit bin name
(5,7,9,11)	s_5_7_9_11
(15:37)	s_15_37
(base*10+3)	s_base_10_3
(all)	s_all

Conditionals

You can add conditional statements at the end of any state declaration. Conditional statements can be any valid Vera conditional statement. Functions other than `get_cycle()` and `get_time()` cannot be called in the conditional. If there is a conditional statement attached to the state declaration, the bin counter will be incremented only if both the condition is true and the state variable matches the state specification at the same time. For example:

```
state jmp_ins (8'b0000_01XX, 8:10, 15:17:7:2) if (test == ON);
```

This state declaration creates the bin `jmp_ins`. The bin counter is incremented when the state variable matches the specification **AND** the conditional is true.

Multiple State Bin Declarations

A state declaration can have multiple bins declared on a single line as follows:

```
state b0 (0), b1 (1), b2 (2);
```

In this example, `b0`, `b1`, and `b2` are separate states, each with its own state specification.

Further, multiple state declarations are allowed, and the same state can be associated with multiple bins. For example:

```
state b0 (0:10);
state b1 (10:20);
```

In this example, state 10 is associated with both bin `b0` and bin `b1`.

m_state

The `m_state` state declaration is used to declare multiple state bins up to a maximum of 4096 bins. The syntax is:

```
m_state state_bin_name (exp1:exp2);
```

state_bin_name - The `state_bin_name` is the base name of the state bins being created.

exp - The `exps` can be any valid coverage expression. You cannot call functions in the expressions. The expressions can include variables.

When the `m_state` declaration is used, multiple state bins are created, covering all the values in the range. The expressions are evaluated when the coverage object is instantiated. For example:

```
m_state s1 (2:4);
```

This example creates these bins with their respective state values:

state bin name	state value
s1_0000	2
s1_0001	3
s1_0002	4

If no bin name is specified, the same example yields these bin names and values:

state bin name	state value
s_0002	2
s_0003	3
s_0004	4

11.2.2.2 Illegal State Declarations

Illegal state declarations associate illegal states with a monitor bin. The syntax is:

```
bad_state error_bin_name (state_specification) conditional;
```

Illegal or bad states are those states in the design that, when entered, result in verification errors.

The state specification can be any expression or combination of expressions as in the state declarations. However, it is often useful to define every state that is not in the state declarations as a bad state. To use that definition of bad states, you can use the `not state` specification:

```
bad_state error_bin_name (not state);
```

This statement increments the specified bin counter every time the state variable matches a value not defined in the state declarations. If you do not specify an error bin name, the implicit name is `s_not_state`.

If you want to specify multiple bad states, you can use the `m_bad_state` declaration. The syntax is:

```
m_bad_state error_bin_name (exp1:exp2);
```

When the `m_bad_state` declaration is used, a bin for each value in the range is created. If no bin name is specified, the same naming conventions for `m_state` are used.

11.2.2.3 Transition Declarations

Transition declarations associate state transitions with monitor bins. The syntax for transition declarations is:

```
trans trans_bin_name (state_transitions) conditional;
```


State Transition

State transitions are specified by declaring a sequence of transitions between states. The general format is:

```
trans trans_bin_name (state_set_1 -> state_set_2 -> ... -> state_set_N);
```

A state set is made up of one or more transition states. Transition states can be any of the following:

1. a state bin, specified in a state declaration
2. any of the state specifiers available in a state declaration
3. a Perl regular expression matching a state bin name, enclosed in double quotes

If multiple transition states make up a single state set, they must be enclosed by brackets ([]) and separated by commas. For example:

```
trans trans_bin_name ([jms_ins, br:xor, "jmp[0-9]+" ] -> [15:20:2:100]);
```

This example defines a complex state transition. The specified bin counter is incremented when any of the following occurs:

- (i) the state variable changes from a state defined in the jms_ins state declaration to a state that falls in the repeated range defined by [15:20:2:100]
- (ii) the state variable changes from a value in the range from br to xor to a state that falls in the repeated range defined by [15:20:2:100]
- (iii) the state variable changes from a state defined in any state bin that begins with "jmp" and ends with at least one digit to a state that falls in the repeated range defined by [15:20:2:100]

A special case for transition states is the all state transition argument.

```
trans trans_bin_name (all);
```

The specified bin counter is incremented on any state variable transitions.

Note – If you use `trans(all)`, a significant amount of memory is used if the number of sampled and unique transitions is large.

Transition Bin Names

Bins can be assigned explicit names, or the Vera compiler can generate implicit names. The Vera compiler generates implicit bin names based on the state transition. The general format is always prefaced by `t_`. The following characters are converted to underscores (`_`): comma (`,`), colon (`:`), slash (`/`), dash (`-`), plus (`+`), asterisk (`*`), and period (`.`). The following characters are ignored when the bin name is generated: double quote (`"`), brackets (`[]`), angled brackets (`<>`), parentheses (`()`), dollar sign (`$`), caret (`^`), backslash (`\`), and braces (`{}`). For example:

transition expression	implicit bin name
(5 -> 7 -> 9 -> 11)	t_5_7_9_11
(15:37 -> 2:5)	t_15_37_2_5
((base*10, 9] -> "jmp[0-9]+")	t_base_10_9_jmp0_9_
(all)	t_all

Conditional

You can add conditional statements at the end of any transition declaration. Conditional statements can be any valid Vera conditional statement. Functions other than `get_cycle()` and `get_time()` cannot be called in the conditional. If there is a conditional statement attached to the transition declaration, the bin counter will be incremented only if both the condition is true and the state variable makes the specified transition at the same time. For example:

```
trans jmp_ins (8:10 -> 15:17:7:2) if (test == ON);
```

This transition declaration creates the bin `jmp_ins`. The bin counter is incremented when the state variable makes the specified transition **AND** the conditional is true. If you specify a sequence of transitions, the conditional is only evaluated during the *final* transition.

Multiple Transition Bin Declarations

A transition declaration can have multiple bins declared on a single line as follows:

```
trans b0 (0 -> 1), b1 (1 -> 2), b2 (2 -> 3);
```

In this example, `b0`, `b1`, and `b2` are separate transitions, each with its own transition specification.

A special case of a transition declaration uses the `all` transition specification.

```
trans trans_bin_name (all);
```

This statement indicates that all transitions increment the specified bin counter.

m_trans

The `m_trans` transition declaration is used to declare multiple transition bins up to a maximum of 4096 bins. The syntax is:

```
m_trans trans_bin_name (exp1:exp2 -> exp3:exp4);
```

trans_bin_name - The `trans_bin_name` is the base name of the transition bins being created.

exp - The `exps` can be any valid coverage expression. You cannot call functions in the expressions. The expressions can include variables.

When the `m_trans` declaration is used, multiple transition bins are created, covering all the transitions in the specified ranges. Each set of expressions specifies a range. A bin is created for each permutation of valid states. For example:

```
m_trans t1 (2:3 -> 4:5);
```

This example creates these bins with their respective transitions:

transition bin name	transition
t1_0000_0000	2 to 4
t1_0000_0001	2 to 5
t1_0001_0000	3 to 4
t1_0001_0001	3 to 5

In this example, states 2 and 3 are given the identifiers 0000 and 0001, and states 4 and 5 are given the identifiers 0000 and 0001. These identifiers are used in the transition bin's implicit name. If other states are added to either side of the transition, they are assigned identifiers in sequential order (i.e., the next state is 0002, and so on).

If no bin name is specified, the same example yields these bin names and values:

transition bin name	transition
t_0000_0000	2 to 4
t_0000_0001	2 to 5
t_0001_0000	3 to 4
t_0001_0001	3 to 5

11.2.2.4 Illegal Transition Declarations

Illegal transition declarations associate an illegal transition with a monitor bin. The syntax is:

```
bad_trans trans_bin_name (state_transitions) conditional;
```

The state transition can be any state transition set valid for transition declarations. However, it is often useful to monitor all transitions that have not been defined as legal transitions. For such instances, Vera uses the `not trans` argument.

```
bad_trans trans_bin_name (not trans);
```

The counter associated with the specified bin will be incremented every time a transition occurs that is not explicitly defined in the transition declaration. If you do not specify the bin name, the implicit name is `t_not_trans`.

The `not trans` modifier applies only to single transitions between two state sets. It does not apply to sequences of transitions. So, if you use `not trans` and you specify a sequence of transitions (`A -> B -> C`), you must also specify each valid transition within the sequence (`A -> B`, `B -> C`).

If you want to specify multiple bad transitions, you can use the `m_bad_trans` declaration. The syntax is:

```
m_bad_trans error_bin_name (exp1:exp2 -> exp3:exp4);
```

When the `m_bad_trans` declaration is used, a bin for each transition is created. If no bin name is specified, the same naming conventions for `m_trans` are used.

11.2.2.5 Coverage Declaration Example

```
state s_IDLE ( IDLE ), s_FLUSH_C( FLUSH_C ), s_LOAD_C ( LOAD_C );
state s_B_RD ( B_RD0:B_RD3 ), s_B_WR ( B_WR0:B_WR3 );
state s_all_RD ( S_RD, B_RD0:B_RD3 ), s_all_WR ( S_WR, B_WR0:B_WR3 );
bad_state ( not state );

trans t_Noop ( IDLE -> IDLE ) if ( mbind.$cmd === NO_OP);
trans t_CacheMissDirty ( IDLE -> "s_FLUSH_C" -> LOAD_C -> START )
    if ( mbind.$cmd !== NO_OP && mbind.$cmd !== FLUSH_CACHE
        && mbind.$cache_hit === 0 && mbind.$cache_dirty === 1 );
trans t_SingleRead (START -> S_RD) if (mbind.$cmd === SINGLE_READ );
trans t_SingleWrite (START -> S_WR) if (mbind.$cmd === SINGLE_WRITE);
trans t_BurstWrite(START -> B_WR0 -> B_WR1 -> B_WR2 -> B_WR3)
    if(mbind.$cmd === BURST_WRITE);
```

This example is a small piece of a complete coverage declaration. This example declares many valid states under various bin names. It also declares several valid transitions. Each time one of the defined activities occurs, the counter to the corresponding monitor bin is incremented by one.

11.2.3 Measure of Coverage

The measure of coverage defines how coverage is specified. The default setting for the measure of coverage computes the percentage of monitor bins that have a non-zero count. The measure of coverage is recomputed each time `query(GOAL)` is called or each time a coverage report is generated.

You can explicitly define the measure of coverage. The syntax is:

```
coverage_val = 100*query(command, bin_type, "bin_pattern"; operand, hit)
               / query(command, bin_type, "bin_pattern");
```

command - *Command* determines the type of results you are going to query. It can be either `NUM_BIN` or `SUM`. Use `NUM_BIN` to count the number of selected bins. Use `SUM` to calculate the sum of counter values on the selected bins.

bin_type - The *bin_type* specifies the bin types you want to select from. The bin types are `STATE`, `TRANS`, `BAD_STATE`, and `BAD_TRANS`. To select more than one type of bin, use the 'or' operator (`|`).

bin_pattern - The *bin_pattern* variable is matched against the bin names of the specified type. Only those bins whose names contain the *bin_pattern* are included in the query. For example, if *bin_pattern* is "bus", all bins with "bus" in their names will be included. If you

want to select all bins whose names begin with a specific string, insert a caret (^) before the string (e.g. "^bus"). To select all bins of the specified type regardless of name, use "." as the *bin_pattern*.

operand - The *operand* must be one of the following: greater than (GT), greater than or equal to (GE), less than (LT), less than or equal to (LE), equal to (EQ), or not equal to (NE).

hit - The *hit* specifies the number of counter hits to which the query is compared using the *operand*. It can be any non-negative integer.

The measure of coverage calculates your current coverage. Put simply, it is a ratio of the number of states that have been "hit" to the total number of valid states.

Using these definitions, you can explicitly define the measure of coverage. For example, to determine the percentage of STATE and TRANS bins with at least one counter hit, use:

```
coverage_val = 100 * query(NUM_BIN, TRANS|STATE, ".*", GT, 0) /
               query(NUM_BIN, TRANS|STATE, ".*");
```

This example is the default measure of coverage.

11.2.4 Coverage Goal

The syntax to set the coverage goal is:

```
coverage_goal = expression;
```

expression - The *expression* can be any valid coverage block expression.

The coverage goal is a percentage between 0 and 100. If there is no explicitly defined goal, the default is 90. The coverage goal should be set such that when it is reached, all coverage objectives have been met.

The coverage goal is most often used with the `query()` system function, discussed in Section 11.4.

11.2.5 Coverage Options

The syntax to set the coverage option is:

```
coverage_option = option;
```

option - The *options* are HI, which specifies that this object will be used for cross-correlation and needs a time stamp, LO, which specifies that only counter information must be retained, or CROSS_TRANS, which includes transition events in a cross coverage report.

Multiple options can be selected using a + or ! to separate options. The default setting is LO. If both HI and LO are chosen, the option is HI. Because the CROSS_TRANS option is used for cross correlation, it can only be used with the HI coverage option.

The **CROSS_TRANS** modifier includes transition events in a cross coverage report. By default, cross coverage reports include only state information. If the **CROSS_TRANS** modifier is active, the cross coverage report contains both state and transition information. In this way, you can include information on transitions that occur when specified states are in use. However, transition/transition associations are not included in the cross coverage report.

Note – If **CROSS_TRANS** is active, it must be active for every coverage block in the coverage report. If it is not, a warning message is issued and the cross coverage report is not generated.

11.2.6 Initialization

Coverage definitions are initialized by an embedded case block that is invoked only when the definition is instantiated. The outer level of the case block resembles a standard Vera case statement except that it can also include the full range of coverage statements. The initialization case block can contain any of these elements:

- coverage declarations (state, bad state, transition, and bad transition declarations)
- measure of coverage
- coverage goal
- other case blocks

The syntax for an initialization sequence generally follows this format:

```
case (initialization_paramater)
{
    expression1:statement1;
    expression2:statement2;
    ...
    default:
}
```

Initialization sequences are generally used to conditionalize coverage statements so that the coverage definition is more specialized upon instantiation. For example, an initialization sequence can be particularly useful when passing flags as arguments when the coverage definition is instantiated. For instance, if **state_flag** is defined such that false turns on bad states and true turns off bad states, the following syntax would be used:

```
case (state_flag)
{
    0: bad_state bad_st_bin (0, 2, 3);
    default:
}
```

This example declares states 0, 2, and 3 as bad states if the flag is set. Each time this coverage definition is instantiated, the flag will be checked. This enables you to define one coverage definition, but use it in different ways throughout the simulation.

11.3 Instantiation and Triggering

Coverage definitions are not created until they are instantiated using the `newcov` command. The syntax to instantiate a coverage definition is:

```
coverage_definition instance_name = newcov(state_var, trigger, opt_args);
```

coverage_definition - *Coverage_definition* is simply the name of the coverage definition that you are instantiating.

instance_name - The *instance_name* is the name of this instance of the coverage definition. Remember that `newcov()` can be used to create multiple instances of the same definition.

state_var - The *state_var* is the value that is monitored by the coverage definition. It is the state variable or expression for which the coverage definition has been created.

trigger - The *trigger* is a trigger expression that allows you to control when the object takes a sample. Coverage objects can be triggered on clock edges, variable changes, and sync events.

opt_args - The *opt_args* are the optional arguments. They must correspond to the coverage definition's optional arguments, if any exist.

The trigger can be used to sample on clock edges as per the synchronization command (see Section 4.4.1, "Synchronization"). When the specified clock edge occurs, the object is triggered.

The trigger can also be used to take samples only on changes to variables using the `wait_var` construct see Section 6.1.1.2, "`wait_var()`"). When a variable change occurs, the object is triggered.

Coverage objects can also be triggered on sync events (see Section 6.2.2, "Sync"). When the sync is unblocked the object is triggered.

There are a few things to note when using triggers. First, to eliminate race problems, if the trigger involves clock delays, the coverage object is invoked at the end of the simulation cycle after all Vera processing that can change variables is complete. Second, if the sample occurs on every change, the coverage object is triggered at most once per simulation cycle even if the state variable changes multiple times.

```
program memCtl_Test
{
    t_CmdPacket CmdPacket;
    t_MemCtl_Cov MemCtl_Cov;
    t_Cmd_Cov Cmd_Cov;
    event evt_new_cmd;
```

```

// instantiates global objs
reset_mc with mbind ();
fork
{state_snooper with mbind();}
{cmd_snooper ();}
join none

// Test with GENERIC_COV value
CmdPacket = new;
CmdPacket.command = NO_OP; // just to initialize before coverage begins
MemCtl_Cov=newcov(mbind.$mc_state,@(posedge CLOCK),HI, 100,
    GENERIC_COV );
Cmd_Cov=newcov(CmdPacket.command, sync ( ALL, evt_new_cmd ) );
printf ("##### GENERATE SEQUENCE FOR GENERIC COVERAGE #####\n\n");
generate_test with mbind ( );
coverage ( CROSS, MemCtl_Cov, Cmd_Cov, "generic.rpt" );
} //end program

task reset_mc with mport (){
    $cmd = NO_OP async;
    @2 $mc_state == IDLE; } //end reset_mc

task state_snooper with mport ()
{
    string s_str;
    while (1)
    {
        @(posedge CLOCK);
        case ( $mc_state )
        {
            4'd0 : s_str = "IDLE";
            4'd1 : s_str = "START";
            4'd2 : s_str = "FLUSH_C";
            default : s_str = "UNKNOWN_STATE";
        }
        //end case
        if ( $mc_state == 4'd0 ) printf (" \n");
        printf (" %s -> ", s_str);
    } //end while
} //end task state_snooper

task cmd_snooper ()
{
    string c_str;
    while (1)
    {
        sync (ALL, evt_new_cmd );
    }
}

```



```

case ( CmdPacket.command )
{
    3'd0 : c_str = "NO_OP";
    3'd1 : c_str = "BURST_READ";
    3'd2 : c_str = "BURST_WRITE";
    3'd3 : c_str = "SINGLE_READ";
    default: error ("Unknown Command %0d\n", CmdPacket.command);
}
printf (" ## CMD : %s ## ", c_str );
delay ( 2 );
} //end while
} //end cmd_snooper;

```

This example forks off two processes, each of which calls one of the user-defined tasks. The `cmd_snooper()` task calls a `sync`, which triggers one of the coverage objects. The other coverage object is triggered on the positive clock edge.

11.4 Coverage Feedback: The Query Function

There are a set of related, pre-defined methods associated with coverage objects that allow you to review the current coverage status. These are all integer functions with the name `query`. The query functions are in these forms:

```

query(command);
query(command, bin_type);
query(command, bin_type, bin_pattern);
query(command, bin_type, bin_pattern, operand, hit);

```

command - The basic form of query allows you to specify one of several *commands*:

NUM_BIN, SUM, FIRST, NEXT, GOAL and NAME. Each command returns a different value based on the *bin_type*.

<u>Command</u>	<u>Function</u>
NUM_BIN	Counts the number of bins of type <i>bin_type</i>
SUM	Sums the counters for all bins of type <i>bin_type</i>
FIRST	Returns the count of the first bin of type <i>bin_type</i> (or -1 for failure)
NEXT	Returns the count of the next bin of type <i>bin_type</i> in the sequence started with FIRST (or -1 if the sequence is complete)
GOAL	Returns a 1 if the <i>coverage goal</i> has been met (0 otherwise)

Note - NEXT and GOAL can be used only in the single-argument form.

The single-argument form of the command applies to all bins. The additional arguments can be added to narrow the bin selection.

bin_type - The *bin_type* is any of the valid bin types: STATE, BAD_STATE, TRANS and BAD_TRANS. Multiple bin types can be specified using the 'or' operator (!).

bin_pattern - The *bin_pattern* variable is matched against the bin names of the specified type. It can be any regular Perl expression. Only those bins whose names contain the *bin_pattern* are included in the query. For example, if *bin_pattern* is "bus", all bins with "bus" in their names will be included. If you want to select all bins whose names begin with a specific string, insert a caret (^) before the string (e.g. "^bus"). To select all bins of the specified type regardless of name, use ".*" as the *bin_pattern*.

operand - The *operand* must be one of the following: greater than (GT), greater than or equal to (GE), less than (LT), less than or equal to (LE), equal to (EQ), or not equal to (NE).

hit - *Hit* specifies the number of counter hits to which the query is compared using the *operand*. It can be any non-negative integer.

So, to compute the sum of the bin counts for state or transition bins with counts greater than 10 each, we would have:

```
query(SUM, STATE|TRANS, ".*", GT, 10);
```

To continue a process until at least 10 bins has 10 or more hits, use:

```
while (mycovobj.query(SUM, STATE|TRANS, ".*", GT, 10) < 11)
{...}
```

You can also set up loops that continue processes until the coverage goal is met:

```
while (!mycovobj.query(GOAL))
{...}
```

It is also possible to query the name of a monitor bin. The syntax is:

```
query_str(NAME);
```

This function returns the monitor bin name in the FIRST/NEXT series (or null if the series is complete).

Note - Outside of a coverage declaration we need to qualify query to get the appropriate method. For example, to get the query method for the cp0 coverage object, we would use cp0.query.

11.5 Coverage Administration

These coverage tasks provide administrative control of coverage objects. The general format is:

```
coverage(command [, object_list] [, "filename"]);
```

command - The *commands* are:

<u>Command</u>	<u>Function</u>
ON	Turns coverage on (enables bin updating)
OFF	Turns coverage off (disables bin updating)
CLEAR	Clears all monitor bin counters
SAVE	Writes the bin data to the specified file
LOAD	Reads the bin data from the specified file
REPORT	Generates a coverage report
CROSS	Generates a cross-correlation report

object_list - The *object_list* lists the coverage objects to be included in the execution of the *command*. If it is not included, all coverage objects will be included.

filename - The *filename* specifies where the cross coverage report is saved. It is also used to specify the read and write files for the **SAVE** and **LOAD** commands.

These commands allow you general control of coverage objects. They also allow you to do incremental coverage of your tests. At the beginning of a new test, you can **LOAD** the current coverage state, run the test to update the state, and then **SAVE** that state back to a file - to be updated again by a later test.

Note - You generally want to keep coverage turned off until after the circuit has been reset.

The **REPORT** option generates a coverage report, detailing each bin and the number of hits it has taken. The **CROSS** report does a cross correlation between objects. Remember that the coverage option has to be set to **HI** for each selected coverage object to be correlated.

When using **SAVE/LOAD**, coverage objects must be loaded in the same order as they were saved. In the "LOADING" program, coverage definitions must also be instantiated in the same order as they were in the "SAVING" program.

This is an example call for a coverage report:

```
coverage (REPORT, memCtl_cov, "memCtl.rpt");
```

This is the resultant coverage report:

```
Coverage Instance Name: MemCtl_Cov
Coverage Instance Handle: 3
Coverage Definition Name: t_MemCtl_Cov
Coverage Goal: 1
Coverage Option: 1
Measure of Coverage: 1
Coverage Sample Expression:
```

State Name	# defined values	# hits
-----	-----	-----
s_START	1	125
s_B_RD0	1	52
s_B_RD1	1	52
s_B_RD2	1	51
s_B_RD3	1	52
s_B_WR0	1	35
s_B_WR1	1	35
s_B_WR2	1	35
s_B_WR3	1	35
s_not_state	0	0

Transition Name	# defined transitions	# hits
-----	-----	-----
t_s_0_s_1	1	59
t_s_0_s_3	1	31
t_s_3_s_1	1	66
t_s_2_s_3	1	35
t_s_0_s_2	1	610
t_s_2_s_0	1	575
t_s_10_s_11	1	35
t_s_11_s_12	1	35
t_not_trans	0	0

11.6 Cross Coverage

Cross coverage refers to the timestamping of events to ensure that specified events happen simultaneously. For instance, a transition from state 1 to state 2 may not be sufficient for testing the device. You may need to know that a transition from state 1 to state 2 occurred while another state was active.

To use cross coverage, you must set the coverage option to HI. You can also optionally set the coverage option to CROSS_TRANS if you want to include transition information in a cross coverage report.

To invoke a cross coverage report, you must invoke the coverage() system function with the CROSS command:

```
coverage(CROSS, object_list, "filename");
```

This is an example of a program using cross coverage:

```
#define OUTPUT_EDGE PRO
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>
#include "counter.if.vrh"

coverage_def cc_bitVec3(bit [2:0] bitVec, integer cov_opt )
{
  /*
    8 states: s_0,...s_7
    7 trans : t_s_0_s_1,...,t_s_6_s_7 and t_s_0_7_s_0
    1 bad_state : s_not_state
    1 bad_trans : s_not_trans
    default coverage_val, coverage_goal
    coverage_option = cov_opt
  */
  state (0), (1), (2), (3), (4), (5), (6), (7);
  trans t_s_0_s_1 (0 -> 1), t_s_1_s_2 (1 -> 2), t_s_2_s_3 (2 -> 3),
    t_s_3_s_4 (3 -> 4), t_s_4_s_5 (4 -> 5),
    t_s_5_s_6 (5 -> 6), t_s_6_s_7 (6 -> 7);
  trans t_s_0_7_s_0 (0:7 -> 0);
  bad_state ( not state );
  bad_trans t_not_trans ( not trans );

  coverage_option = cov_opt;
} //end cc_bitVec3 defn

program counter_test
{ // start of top block
  cc_bitVec3 cc_cnth, cc_cnth2;
  string rptFileCrossSame = "crossCcSame.rpt";

  // Start of counter_test
  reset_counter();

  // start the high coverage , and clear their values
  cc_cnth=newcov(counter.count, @(counter.count async), HI+CROSS_TRANS);
  cc_cnth2=newcov(counter.count, @(counter.count async), HI+CROSS_TRANS);
```

```

repeat (2) @(posedge CLOCK);
coverage ( CLEAR, cc_cnth, cc_cnth2 );
coverage ( CLEAR, cc_cnth2);
repeat (2) @(posedge CLOCK);

// OFF and ON with obj-list to be no arguments
drive_counter ( 4 ); // all cov-counters up by 4
drive_counter ( 5 );

// CROSS coverage report of the 2 virtually similar test
coverage ( CROSS, cc_cnth, cc_cnth2, rptFileCrossSame );
} // end of program counter_test

task reset_counter ()
{
    @0 counter.start = 0;
    counter.stop = 1;
    @2 counter.count == 0;
} //end reset_counter

task drive_stop ( integer num_times )
{
    repeat ( num_times )
    {
        @1 counter.stop = 1;
        @2 counter.count == 0;
    }
} //end drive_stop

task drive_counter ( integer num_times )
{
    integer exp_count, i;
    printf ("Driving counter %0d times...\n", num_times);
    repeat ( num_times )
    {
        @0 counter.count == 3'd0;
        exp_count = 0;
        @1 counter.start = 1;
        for ( i = 0; i < 8; i++ )
            @1 counter.count == exp_count++;
        @0 counter.stop = 1;
        @2 counter.count == 3'd0;
    } //end repeat
} //end drive_counter

```

This example creates a coverage definition that declares 8 legal states and 7 legal transitions. Then the main program instantiates two coverage objects. The counter is driven multiple times and the states and transitions are monitored by the coverage objects. The resulting coverage report is:

cc_cnth	cc_cnth2	# hits
-----	-----	-----
s_0	s_0	9
s_0	t_s_0_7_s_0	9
s_1	s_1	9
s_1	t_s_0_s_1	8
s_2	s_2	9
s_2	t_s_1_s_2	9
s_3	s_3	9
s_3	t_s_2_s_3	9
s_4	s_4	9
s_4	t_s_3_s_4	9
s_5	s_5	9
s_5	t_s_4_s_5	9
s_6	s_6	9
s_6	t_s_5_s_6	9
s_7	s_7	9
s_7	t_s_6_s_7	9
t_s_0_7_s_0	s_0	9
t_s_0_s_1	s_1	8
t_s_1_s_2	s_2	9
t_s_2_s_3	s_3	9
t_s_3_s_4	s_4	9
t_s_4_s_5	s_5	9
t_s_5_s_6	s_6	9
t_s_6_s_7	s_7	9

```
Maximum Coverage Instance bin combinations: 256
Detected Coverage Instance bin combinations: 24
Percentage combinations: 9.375
```

This coverage report lists all of the states for each coverage object, and the number of times the other coverage object observed a state activity or transition activity during the state.

11.7 Backward Compatibility

Vera continues to support an earlier version of functional coverage that was not integrated into the object-oriented framework. In this implementation, you declare a coverage block for each state variable you wish to check. These coverage blocks are then automatically invoked each time the state variable changes.

Coverage blocks need to be declared in the main program, either before the **program** keyword or immediately after:

```
coverage_block name
{
    <vector declaration>
    <state declaration>*
    <transition declaration>*
    <coverage options>
}
```

The vector declaration is a list of Vera input or inout signals as specified in an interface statement. Sampling is done at the edge associated with the first signal.

The state declarations associate a name with a numeric value, defining all of the legal states. The statements are of the form:

```
state name number;
```

The transition declarations come in two forms. Self-transitions are of the form:

```
transition name:name;
```

which says we will allow a transition from *name* to *name*. More general transitions are specified with:

```
transition source: dest1, dest2, dest3;
```

which specifies transitions from *source* to any of the specified destinations.

11.7.1 Coverage Options

The coverage options for the earlier version of functional coverage are:

Coverage option	Effect
illegal_state	Reports when an illegal state is entered
illegal_transition	Reports when an illegal transition occurs
illegal_self_transition	Reports illegal self-transitions when enabled

The default is for all self transitions to be legal. However, if it is not valid to stay in a state for more than one cycle, you can use this option. In this mode, you must specify the self-transitions that are valid. All other self-transitions are reported as illegal.

11.7.2 Coverage Reporting

All of the coverage tasks that were outlined earlier (except CROSS) work with coverage blocks. A call to `coverage(REPORT)` produces a report that details the status for all the specified legal and illegal transitions:

```
**** VERA Coverage Report for fsm_check ****
<src>      <dst>      <num>      <last_cycle>
IDLE       PEND      5         5428
IDLE       WR0       22        4543
IDLE       RD0       24        3854
PEND       IDLE      0         -
.
.
```

VERA COVERAGE ERROR: PEND -> IDLE transition is not covered

11.7.3 Conditional Coverage

Sometimes it is necessary to have finer visibility into the state-machine behavior than just which states and transitions were taken. In this case, adding pseudo-states can often provide the requisite refinement. For example, we can cover state transitions driven by a control signal by creating two IDLE states:

```
coverage fsm_check_i {
  vector  fsm.cnt1, fsm.state;
  state   PEND      5'bx0001; // first bit does not affect state
  state   IDLE_Z     5'b00000;
  state   IDLE_O     5'b10000;
  trans   IDLE_Z,    PEND; // transition while cnt1==0
  trans   IDLE_O,    PEND; // transition while cnt1==1
}
```

We check for transitions from the IDLE to the PEND state, but now for both possible values of the fsm.cnt1 signal.

Copyright © 2004 Synopsys, Inc.

Example 6: Coverage Analysis

This is an example using functional coverage analysis. The example includes these features:

- State and transition coverage
- Coverage options
- Coverage values
- Coverage object triggering

This example includes these files:

- *globaldef.h*
- *memCtl.if.vrh*
- *main.vr*
- *coverage.vr*
- *RUN* (a run script)

The *RUN* script generates a header file for *coverage.vr*.

Files included in the distribution but not shown here include:

- *vsg.vrh*
- *stingen.vrg*

globaldef.h

```
#ifndef FILE_GLOBALDEF_H
#define FILE_GLOBALDEF_H

enum MEM_SEG = MEM_SEG0, MEM_SEG1, MEM_SEG2;
enum COVERAGE_VAL_OPT = GENERIC_COV, STATE_COV, TRANS_COV;

#define W_NOOP 1
#define W_FLUSH_CACHE 2
#define W_BURST_READ 8
#define W_SINGLE_READ 4
#define W_BURST_WRITE 7
#define W_SINGLE_WRITE 3

#endif
```

memCtl.if.vrh

```

#ifndef INC_MEMCTL_IF_VRH
#define INC_MEMCTL_IF_VRH
#define INPUT_EDGE PSAMPLE
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1

interface memCtl
{
    input [3:0] mc_state INPUT_EDGE ;
    inout [2:0] cmd INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW ;
    inout cache_hit INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW ;
    inout cache_dirty INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW ;
    input clk CLOCK ;
} // end of interface memCtl

port mport
{
    mc_state;
    cmd;
    cache_hit;
    cache_dirty;
} //end mport

bind mport mbind
{
    mc_state memCtl.mc_state;
    cmd memCtl.cmd;
    cache_hit memCtl.cache_hit;
    cache_dirty memCtl.cache_dirty;
} //end bind

#endif

```

main.vr

```

#include <vera_defines.vrh>
#include "RTL/memCtl.vh"
#include "memCtl.if.vrh"
#include "coverage.vrh"
#include "stimgen.vrh"
#include "vsg.vrh"

program coverage_example
{
    t_Cmd_Cov    Cmd_Cov;

```

```

t_MemCtl_Cov MemCtl_Cov;
t_CmdPacket  CmdPacket;
event        evt_new_cmd;

// instantiate command packet

CmdPacket = new;
CmdPacket.command = NO_OP; // just to initialize before coverage begins

// instantiate coverage objs

MemCtl_Cov = newcov (mbind.$mc_state,@(posedge CLOCK), HI,
                    100, GENERIC_COV);
Cmd_Cov = newcov ( CmdPacket.command, sync ( ALL, evt_new_cmd ) );

printf ( "***** GENERATE SEQUENCE FOR GENERIC COVERAGE
          *****\n\n" );
generate_test with mbind ( );
coverage ( CROSS, MemCtl_Cov, Cmd_Cov, "generic.rpt" );
} //end program

```

coverage.vr

```

#include <vera_defines.vrh>
#include "RTL/memCtl.vh"
#include "memCtl.if.vrh"
#include "stimgen.vrh"

extern t_CmdPacket CmdPacket;

// Coverage Obj Type t_MemCtl_Cov
// cover the command state-machine-values
// and the transitions among them
// single-transitions are listed to prevent bad_trans (not trans)
// from catching transitions defined in multiple-trans bins
// but not covered individually

// coverage_val is calculated by 3 types of coverage-levels controlled
// by argument 4

coverage_def t_MemCtl_Cov (bit [3:0] mc_state, integer cov_opt,
                          integer cov_goal, COVERAGE_VAL_OPT cov_val_opt )
{
    state s_IDLE ( IDLE ), s_FLUSH_C( FLUSH_C ), s_LOAD_C ( LOAD_C ),
        s_START ( START ), s_B_RD0 ( B_RD0 ), s_B_RD1 ( B_RD1 ),
        s_B_RD2 ( B_RD2 ), s_B_RD3 ( B_RD3 ), s_S_RD ( S_RD ),
        s_S_WR ( S_WR ), s_B_WR0 ( B_WR0 ), s_B_WR1 ( B_WR1 ),
        s_B_WR2 ( B_WR2 ), s_B_WR3 ( B_WR3 );

```

```

state s_B_RD ( B_RD0:B_RD3 ), s_B_WR ( B_WR0:B_WR3 );

state s_all_RD ( S_RD, B_RD0:B_RD3 ), s_all_WR ( S_WR, B_WR0:B_WR3 );

bad_state ( not state );

trans t_Noop ( IDLE -> IDLE ) if ( mbind.$cmd === NO_OP),
  t_CacheHit ( "s_IDLE" -> "s_START" )
    if ( mbind.$cmd !== NO_OP && mbind.$cmd !== FLUSH_CACHE
      && mbind.$cache_hit === 1 ),
  t_CacheMiss ( "s_IDLE" -> LOAD_C -> "s_START" ) if ( mbind.$cmd !==
    NO_OP && mbind.$cmd !== FLUSH_CACHE && mbind.$cache_hit === 0
    && mbind.$cache_dirty === 0 ),
  t_CacheMissDirty ( IDLE -> "s_FLUSH_C" -> LOAD_C -> START )
    if ( mbind.$cmd !== NO_OP && mbind.$cmd !== FLUSH_CACHE
    && mbind.$cache_hit === 0 && mbind.$cache_dirty === 1 ),
  t_FlushCache ( IDLE -> FLUSH_C -> IDLE ) if ( mbind.$cmd ===
    FLUSH_CACHE ),
  t_BurstRead ( START -> B_RD0 -> B_RD1 -> B_RD2 -> B_RD3 )
    if ( mbind.$cmd === BURST_READ ),
  t_SingleRead ( START -> S_RD ) if ( mbind.$cmd === SINGLE_READ ),
  t_SingleWrite ( START -> S_WR ) if ( mbind.$cmd === SINGLE_WRITE ),
  t_BurstWrite ( START -> B_WR0 -> B_WR1 -> B_WR2 -> B_WR3 )
    if ( mbind.$cmd === BURST_WRITE );

// All Valid Single Trans

trans ( IDLE -> START ), ( IDLE -> LOAD_C ), ( LOAD_C -> START ),
  ( FLUSH_C -> LOAD_C ), ( IDLE -> FLUSH_C ), ( FLUSH_C -> IDLE ),
  ( START -> B_RD0 ), ( B_RD0 -> B_RD1 ), ( B_RD1 -> B_RD2 ),
  ( B_RD2 -> B_RD3 ), ( B_RD3 -> IDLE ), ( S_RD -> IDLE ),
  ( S_WR -> IDLE ), ( START -> B_WR0 ), ( B_WR0 -> B_WR1 ),
  ( B_WR1 -> B_WR2 ), ( B_WR2 -> B_WR3 ), ( B_WR3 -> IDLE );

bad_trans ( not trans );

coverage_option = cov_opt ;
case ( cov_val_opt )
{
  GENERIC_COV :
    coverage_val = query ( NUM_BIN, TRANS|STATE, ".*", GT, 0 )
    / query ( NUM_BIN, TRANS|STATE, ".*" );
  STATE_COV : coverage_val = query ( NUM_BIN, STATE, ".*", GT, 2 )
    / query ( NUM_BIN, STATE, ".*" );
  TRANS_COV :
    coverage_val = query ( NUM_BIN, TRANS, ".*", GT, 2 )
    / query ( NUM_BIN, TRANS, ".*" );
}

```

```

    }//end case
    coverage_goal = cov_goal / 100; // in fraction
  }//end t_MemCtl_Cov

  // Coverage Obj Type t_Cmd_Cov
  // covers the commands and some selected transitions between them
  coverage_def t_Cmd_Cov ( bit [2:0] cmd_value )
  {
    state s_NO_OP ( NO_OP ), s_BURST_READ ( BURST_READ ),
      s_SINGLE_READ ( SINGLE_READ ), s_BURST_WRITE ( BURST_WRITE ),
      s_SINGLE_WRITE ( SINGLE_WRITE ), s_FLUSH_CACHE ( FLUSH_CACHE );

    bad_state ( not state );

    trans back_to_back_burst_rd ( BURST_READ -> BURST_READ ),
      back_to_back_burst_wr ( BURST_WRITE -> BURST_WRITE ),
      t_BURST_READ_BURST_WRITE ( BURST_READ -> BURST_WRITE ),
      t_BURST_WRITE_BURST_READ ( BURST_WRITE -> BURST_READ );
    trans t_ALL_WRITE_ALL_READ ( [BURST_WRITE, "s_SINGLE_WRITE" ]
      -> [ "s_BURST_READ", SINGLE_READ ] );

    coverage_option = HI; //needed to do cross coverage
  }//end t_Cmd_Cov

```

RUN

```

#!/bin/csh -f
rm -f *.vro core *.vshell >& /dev/null
vera -cmp -g -h coverage.vr
if ($status) exit 1
vera -cmp -g main.vr

```


12. Multiple Module Support

This chapter introduces Vera's multiple module support. It describes the different components of multiple module support and details how they are used. This chapter includes these sections:

- Introduction
- Overview
- Project Files
- Configuration Files
- Generating a Project-based Configuration

Note – Multiple modules are not yet supported for VHDL.

12.1 Introduction

Thus far, we have described how to create Vera testbenches as monolithic, single programs that are connected to the simulation through one of two methods: the test-top method and the HDL node method. In addition to these methods, Vera supports a project-based configuration methodology that allows you to create testbenches with multiple modules.

Using the test-top method, the Vera shell is instantiated as a component in a test-top file that was either generated by the Vera template generator or created manually. This method has the advantage of keeping the Vera test-bench source code independent of the actual connection to the simulation, which allows you to use the test-bench in multiple simulation environments without having to recompile the code. However, the test-top method forces you to have a test-top level on which *all* connections between the test-bench and simulation must be made. This limitation is significant when making connections deep in the hierarchy of the simulation structure.

The HDL node method uses constructs in the interface files to connect directly to the simulation environment without the need for a test-top file. This method allows you to easily make connections between the test-bench and the simulation environment, even at points deep in the hierarchy of the simulation structure. However, using the HDL node method forces you to use test-bench source code that is dependent on the connection to the simulation. Therefore, you must recompile the source code each time the connection is changed.

Vera's implementation of project-based configuration grants you the advantages of both earlier alternatives without the limitations associated with each. Project-based configuration allows you to create test-benches that are independent of the actual connections to the simulation environment and connect those benches to any point in the simulation hierarchy without the need for a test-top configuration.

Project-based configuration has the added advantage that Vera test-benches can be treated as modules that can be instantiated multiple times and run concurrently in a single simulation.

The power offered through the project-based methodology is substantial even for users who prefer a monolithic approach, and it can be used without the multiple program capabilities.

12.2 Overview

This section gives a brief overview of the components of the project-based configuration approach.

Vera's multiple module support is based on its implementation of project-based methodology. There are three major components to this approach:

1. An overall project (.proj) file specifying testbenches used in the simulation
2. A configuration (.vcon) file for each testbench module
3. The Vera object (.vro) files for each testbench module type

Project Files

The project file specifies which modules are used in the simulation. For each testbench module being used, the project file must have a corresponding section that specifies the module name, the configuration file, and the Vera object files associated with that particular module.

Configuration Files

Each testbench module must have a corresponding configuration file. The configuration file specifies how the module is connected to the simulation. It connects Verilog signals to Vera interfaces. Configuration files also specify any clocks that are needed, declare all Vera-Verilog task calls, and define clock generators with a specific period and timing.

Note that the connections between the Verilog signals and the Vera interfaces can be changed in the configuration files without recompiling the testbench source code.

Vera Object Files

The Vera object files are binary files created when the source (.vr) files for the testbench modules are compiled. A separate object file is required for each testbench.

12.3 Project Files

The project file contains information for each testbench module being used in the simulation. Each module is listed, followed immediately by a list of the configuration file and object files associated with the module. The syntax for the project file is:

```
main module1_name
    module1_name.vcon
    module1_object1.vro
    ...
    module1_objectN.vro
...

main moduleN_name
    moduleN_name.vcon
    moduleN_object1.vro
    ...
    moduleN_objectN.vro
```

Each module definition begins with the **main** keyword and is followed by a *module_name*. The *module_name* is simply a unique instance name for the module. The definition line is followed by lines specifying the configuration file and all Vera object files associated with that module.

You can include a *module_name.vrl* file in place of the Vera object files.

The project file can contain Vera pre-processor directives and macros such as **#include** and **#define**. The normal Vera style comments (**//** and **/* ... */**) are also supported in the project file.

12.4 Configuration Files

The configuration file contains the information specifying how the testbench module is connected to the simulation. Most importantly, it indicates how signals are connected between the testbench and DUT, and it defines clock generators.

The syntax for a configuration file is:

```
timescale timescale_statement
clock clock_statement
connect connect_statement
veritask veritask_statement
```

Each of these statements is discussed in detail in subsequent sections.

12.4.1 Timescale Statement

The timescale statement sets the time between clock ticks. The syntax is:

```
timescale timescale_description
```

timescale_description - The *timescale_description* is in the form of a normal Verilog timescale.

The timescale statement is optional. If you include a timescale statement in the configuration file, an equivalent Verilog `'timescale` statement is generated in the Vera shell file.

12.4.2 Clock Statement

The clock statement creates a clock generator in the Vera shell file, which drives the specified Vera clock. The syntax is:

```
clock clock_name period period_number offset initial_value
```

clock_name - The *clock_name* is the name of the Vera clock. It is specified as either `SystemClock` or `interface_name.clock_name`.

period_number - The *period_number* specifies the cycle length for the clock in nanoseconds.

offset - The *offset* sets the delay before the clock changes for the first time. The default is 0.

initial_value - The *initial_value* is the value the clock starts at (either 0 or 1). The default is 0.

The clock statement is optional. If a Vera clock does not have a clock statement and is not connected to any Verilog node through a connect statement, it will be listed in the I/O list as an input to the Vera shell module (except for `SystemClock`, which will never be listed as output in the I/O list).

Finally, if a Vera clock has a connect statement, with or without a clock statement, it will not be listed in the I/O list of the Vera shell module.

12.4.3 Clock_alias Statement

The `clock_alias` statement aliases a Vera clock to another Vera clock. The syntax is:

```
clock_alias clock_name1 = clock_name2
```

A clock cannot have a clock statement *and* be on the left-hand side of the `clock_alias` statement. This is disallowed because it assigns two clocks to the same clockname. Such assignments generate compilation errors.

12.4.4 Connect Statement

The connect statement connects a Vera signal or clock to a specified Verilog node. The syntax is:

```
connect direction vera_name verilog_name skew skew_value
```

direction - The *direction* can be one of the following: in, input, out, output, or inout. The in or input direction specifies clocks and signals into Vera from the DUT. The out or output direction specifies clocks and signals out from Vera into the DUT. The inout direction specifies bidirectional signals. The *direction* is checked against the direction specified in the interface. Mismatches result in compilation errors. If the *direction* is not specified, the *direction* is taken from the Vera interface. The *direction* for Vera clocks is ignored.

vera_name - The *vera_name* is the name of the Vera signal you are connecting to the DUT. The format for the *vera_name* is *interface_name.signal_name*, where the interface and signal are specified in the interface specification.

verilog_name - The *verilog_name* is the name of the Verilog signal to which the Vera signal is being connected. It should include the Verilog hierarchical path. If the *verilog_name* is not specified, the Vera signal is not connected directly to a Verilog node. Instead, that signal is assumed to be connected through the I/O port list of the Vera shell module. This is useful if the Vera-shell is explicitly instantiated and connected to the simulation through the module ports. In this case, the port name used would be *interface_name.signal_name*.

skew - The *skew* can be either *iskew* (input skew) or *oskew* (output skew). The default is no skew.

skew_value - The *skew_value* is the length of the skew type specified. If you specify input skew, the *skew_value* should be 0 or negative. If you specify output skew, the *skew_value* should be 0 or positive. The default *skew_value* is 0.

The connect statement is necessary to connect the Vera testbench signals to the DUT.

12.4.5 Veritask Statement

The veritask statement maps a Vera task name to a Verilog task path in a Vera-Verilog task call. The syntax is:

```
veritask vera_task_name = verilog_task_name
```

vera_task_name - The *vera_task_name* is the name of the Vera task being mapped to Verilog.

verilog_task_name - The *verilog_task_name* is the Verilog name of the Vera task. It should include the Verilog hierarchical path.

This statement is optional. If it is missing for a Vera-Verilog task call, a warning is printed and the default (the original path from the veritask statement in the Vera source code) is used.

12.5 Generating a Project-based Configuration

To generate a project-based configuration with multiple modules:

1. Compile the source code for the individual testbench modules.

This generates the Vera object files and Vshell files required to run the simulation with the

testbenches.

2. Create a configuration file for each testbench module.

If the testbench source files are available, you can generate a template configuration file by typing:

```
vera -vcon program.vr
```

program - The *program.vr* contains the program block for the module.

You must then customize the template configuration file as necessary by specifying signal connections, clock periods, and any Verilog tasks that need to be defined in the configuration file. It is important that the signals are connected to the correct Verilog nodes.

3. Create an overall project file that contains information on all modules necessary for the simulation.
4. Create Vera shell files for each testbench module listed in the project file:

```
vera -proj file.proj
```

This generates a Vera shell file called *modulename_shell.v* for each module. This Vera shell file is used during the Verilog compilation. The Vera shell file is generated from the Vshell (.vshell) file created when the testbench source code was compiled.

5. Run the simulation with the +vera_pload switch:

```
verilog +vera_pload=proj_name.proj mod1_shell.v ... modN_shell.v
```

The above flow eliminates the need for a test-top file that instantiates the Vera shell module. However, you can still use the test-top file approach with a project-based approach by deliberately not specifying connections for Vera interface signals in the .vcon file. Unspecified interface signals result in a printed warning message during Vera shell creation. The signals are automatically listed as I/O signals in the Vera shell module. The Vera shell module must then be connected to the rest of the simulation by instantiation.

Note - It is possible to mix the two approaches in a single project file where only some test-bench modules require instantiation.

Example 7: Multiple Modules

This is an example of multiple module usage. This example includes these files:

- *SyncMemMon.vr*
- *SyncMemMon.vcon*
- *SyncMemTest.vr*
- *SyncMemTest.vcon*
- *multimod.proj*
- *RUN* (a run script)

Files included in the example distribution but not present here include:

- *public_tasks.vr*

SyncMem.if.vrh

```
interface SyncMem_if
{
    inout CE OUTPUT_EDGE OUTPUT_SKEW INPUT_EDGE;
    inout OE OUTPUT_EDGE OUTPUT_SKEW INPUT_EDGE;
    inout [7:0] AD OUTPUT_EDGE OUTPUT_SKEW INPUT_EDGE;
    inout [31:0] D INPUT_EDGE PRZ OUTPUT_SKEW;
    input clk CLOCK;
} // end of interface SyncMem
```

globals.vrh

```
#define ADDR_SIZE 8
#define DATA_SIZE 32
```

SyncMemMon.vr

```
#define OUTPUT_EDGE PRZ
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>
#include "public_tasks.vrh"
#include "SyncMem.if.vrh"
```

```

// This program serves as a monitor which runs indefinitely
// and looks for bad signal values (X and Z) on the OE and CE pins
// of the verilog SyncMem model.

// At the same time, it will report any write/read activities to
// the model by monitoring the CE and OE signal combinations

program SyncMem_monitor
{ // start of top block

    // Start of SyncMem_test

    @(negedge SyncMem_if.CE); // wait till initial impedance is overridden
                                // by test or other connections

    while (1)
    {
        @(posedge SyncMem_if.clk);
        SyncMem_if.OE = void; // void drive to synchronize

        fork
        {
            if ( SyncMem_if.CE === 1'b1 && SyncMem_if.OE === 1'b0 )
                rptWriteMem ( "SyncMem_if.AD", SyncMem_if.AD, "SyncMem_if.D",
                               SyncMem_if.D );
            if ( SyncMem_if.OE === 1'b1 )
            {
                @(posedge SyncMem_if.clk);
                rptReadMem ( "SyncMem_if.AD", SyncMem_if.AD, "SyncMem_if.D",
                             SyncMem_if.D );
            }
        }

        {
            if ( SyncMem_if.OE === 1'bx ) rptErrorSig ( "SyncMem_if.OE",
                                                         1'bx );
            if ( SyncMem_if.OE === 1'bz ) rptErrorSig ( "SyncMem_if.OE",
                                                         1'bz );
        }

        {
            if ( SyncMem_if.CE === 1'bx ) rptErrorSig ( "SyncMem_if.CE",
                                                         1'bx );
            if ( SyncMem_if.CE === 1'bz ) rptErrorSig ( "SyncMem_if.CE",
                                                         1'bz );
        }

        join all
    } //end while
} // end of program SyncMem_test

```


SyncMemMon.vcon

```

clock SyncMem_if.clk period 100
connect inout SyncMem_if.CE=SyncMem.CE
connect inout SyncMem_if.OE=SyncMem.OE
connect inout SyncMem_if.AD=SyncMem.AD
connect inout SyncMem_if.D=SyncMem.D
connect output SyncMem_if.clk=SyncMem.clk
// timescale 100ns/10ns
clock SystemClock period 100

```

SyncMemTest.vr

```

#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE

#include <vera_defines.vrh>
#include "globals.vrh"
#include "public_tasks.vrh"
#include "SyncMem.if.vrh"

// This test will write to the verilog SyncMem model
// data = (addr + 5) from addr 0 -> 255
// Then it will read back the data and verify them in
// reverse order of addresses.

program SyncMem_test
{
    integer i;

    // Start of SyncMem_test

    reset_mem();
    printf ("----- setting data -----\\n");
    for ( i = 0; i < 256; i++ )
        set_data ( i, i + 5 );
    printf ("----- getting data -----\\n");
    for ( i = 255; i >= 0; i-- )
        if (get_data(i) != (i+5))
            error ("inconsistent data in memory location %0d getting %0h,
                expecting %0h\\n", i, get_data(i), i+5 );
    printf("----- all done -----\\n");
    exit (1); // to terminate simulation altogether
} // end of program SyncMem_test

```

```

task reset_mem ()
{
    @1 SyncMem_if.OE = 0;
    SyncMem_if.CE = 0;
    SyncMem_if.AD = 8'bZ;
    SyncMem_if.D = 32'bZ;
    @1 SyncMem_if.D = 32'bZ;
} //end reset_mem

task set_data ( bit [ADDR_SIZE-1:0] wrAddr, bit [DATA_SIZE-1:0] wrData )
{
    @1 SyncMem_if.OE = 0;
    @1 SyncMem_if.CE = 1;
    SyncMem_if.AD = wrAddr;
    SyncMem_if.D = wrData;
    @1 SyncMem_if.OE = 0;
    SyncMem_if.CE = 0;
    @1 SyncMem_if.D == void; //void expect
} //end set_data

function bit [DATA_SIZE-1:0] get_data ( bit [ADDR_SIZE-1:0] rdAddr )
{
    @1 SyncMem_if.OE = 1;
    SyncMem_if.AD = rdAddr;
    @(posedge CLOCK);
    get_data = SyncMem_if.D;
    @1 SyncMem_if.OE = 0; //void drive
} //end get_data

```

SyncMemTest.vcon

```

clock SyncMem_if.clk period 100
connect  inout  SyncMem_if.CE=SyncMem.CE
connect  inout  SyncMem_if.OE=SyncMem.OE
connect  inout  SyncMem_if.AD=SyncMem.AD
connect  inout  SyncMem_if.D=SyncMem.D
connect  output SyncMem_if.clk=SyncMem.clk
// timescale 100ns/10ns
clock SystemClock period 100

```

13. Vera-CORE

This chapter introduces Vera-Core and describes how it is used. It includes these sections:

- Introduction
- Vera-CORE Usage Model For IP Vendors
- Vera-CORE Usage Model for IP Users

13.1 Introduction

Developers of ASICs often look outside of their own firm for help in creating complex circuits. They purchase synthesizable modules (IP cores) from IP vendors, and integrate the cores into their RTL descriptions. Using IP cores provides two very important advantages for the developers: it shortens the development time, and it allows them to purchase debugged, robust modules.

However, while offering distinct advantages, there is an element of risk in using IP cores. Because developers are generally not as knowledgeable of others' work, latent bugs in the cores can lead to enormous losses of time. Additionally, bugs can be introduced to the ASIC design by improper connection or use of the cores.

Because these bugs can spell disastrous results for both the vendors and developers, both groups must strive to eliminate such problems. In doing so, the traditional approach has been to include monitors with the IP cores that catch problems and notify developers. The monitors are used to check for protocol violations by the system, to check for protocol violations by the core itself, to check for internal error states within the core, and to accumulate test coverage information at the core boundary and within the core.

Traditionally, these monitors have been embedded in the IP RTL using Verilog code. However, the inherent flaws with Verilog in creating and maintaining complex monitors have made such solutions very limited and incomplete.

While Verilog is generally inadequate for such tasks, Vera is a language explicitly designed for expressing these monitoring functions. Vera's multiple module support, flexible connection scheme, and project-based configuration capabilities make Vera a superior tool for developing such checking mechanisms. Moreover, Vera has the unique capability to support Vera modules written by IP vendors without the need for regular Vera runtime licenses. This means that developers can run simulations using the IP cores (written in Vera) without purchasing Vera licenses by simply connecting the testbench signals to the hierarchical DUT signals where the Vera-CORE is to be stitched.

A general flow diagram of Vera-CORE is shown in Figure 13-1.

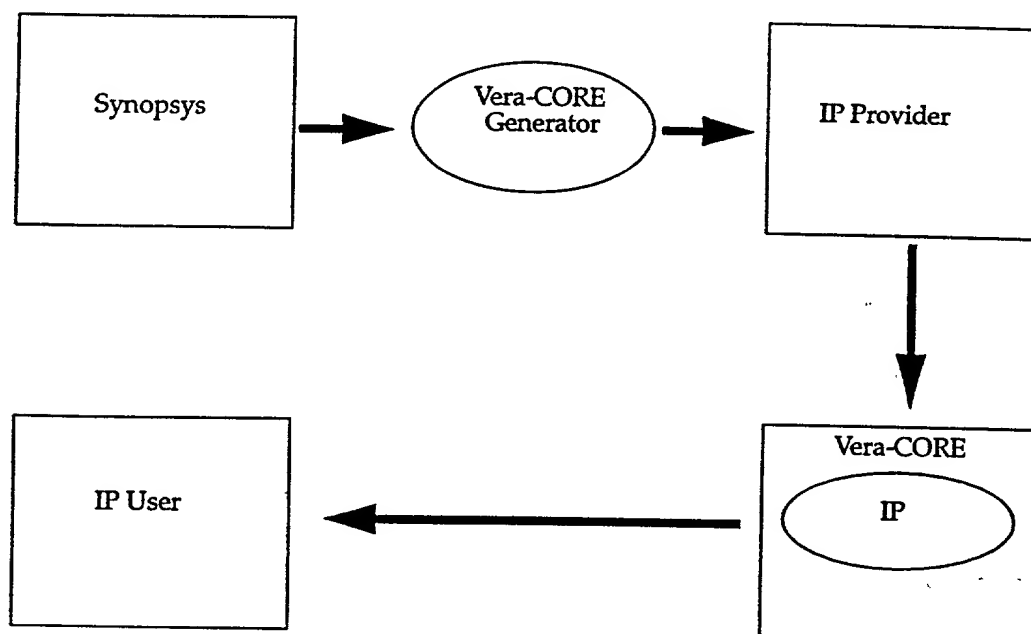


Figure 13-1 General Vera-CORE flow diagram

13.2 Vera-CORE Usage Model For IP Vendors

This section details how IP vendors use Vera to integrate monitors into their IP cores so that developers can use the cores in their simulation environment.

The steps to create the Vera-CORE file set are:

1. Include monitors and coverage objects within the Vera source (.vr) files.
2. Compile the Vera source files using the `-ip` switch. This creates Vera binary files that can be used without a Vera runtime license.

3. Create the configuration file:

```
vera -vcon filename.vr
```

filename - The *filename* is the name of the IP core source file.

4. Create the project file for the IP core including the configuration and object files to be used.

5. Create the Vera shell files:

```
vera -proj filename.proj
```

filename - The *filename* is the name of the IP core project file.

6. Distribute the resultant Vera shell files (shell.v), object files (.vro), configuration files (.vcon), and project file (.proj).

13.3 Vera-CORE Usage Model for IP Users

This section details how IP users use Vera-CORE files without a Vera runtime license.

To use Vera-CORE files:

1. Download the Vera-CORE runtime library from the Synopsys ftp site.
2. Modify the existing configuration files to make direct signal connections between the IP core and the HDL design. Each testbench signal must be connected to the hierarchal signal path where Vera-CORE is to be stitched.
3. Create a Vera list file (.vrl) that includes the project (.proj) and module (shell.v) files.
4. Run the Verilog simulation using this switch:

```
+vera_mload=filename.vrl
```

the first and only time you can see the results of your simulation in a single window.

Example 8: Vera-CORE

This is an example of Vera-CORE usage. This example includes these files:

- *counter.if.vrh*
- *counter_checker.vrh*
- *counter_checker.vr*
- *counter_checker.vcon*
- *counter_checker.vrl*
- *counter_checker.proj*
- *counter_coverage.vrh*
- *counter_coverage.vr*

To generate this Vera-CORE usage model, you should:

1. Generate the test-top file, interface file, and Vera source file:

```
vera -temp -t counter -c clk counter.v
```

This creates the files *counter.test_top.v*, *counter.if.vrh*, and *counter.vr.tmp*.

2. Edit the *counter.if.vrh* interface file to make the signal connections.

3. Generate the configuration (.vcon) file:

```
vera -vcon counter_checker.vr
```

4. Edit the *counter_checker.vcon* file to make the correct signal connections.

5. Instantiate *counter_checker_shell* inside the DUT.

6. Use the RUN script.

counter.if.vrh

```
#ifndef INC_COUNTER_IF_VRH
#define INC_COUNTER_IF_VRH

// Note: counter.if.vrh is edited so that all signals are inputs
interface counter_checker
{
    input rst_n PSAMPLE;
    input clk CLOCK;
```

```

    input Qout PSAMPLE;
    input [3:0] COUNT PSAMPLE verilog_node "counter.count_reg[3:0]";
} // end of interface counter

#endif

```

counter_checker.vrh

```

#ifndef INC__TMP_COUNTER_CHECKER_VRH
#define INC__TMP_COUNTER_CHECKER_VRH
#endif

```

counter_checker.vr

```

#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>
#include "counter.if.vrh"
#include "counter_coverage.vrh"

program Counter_Checker
{
    reg [3:0] count;

    counter_sm_cov cnt0 =
        newcov(counter_checker.COUNT,@(counter_checker.clk));
    while(1)
    {
        if(counter_checker.rst_n == 1)
        {
            printf("counter_checker.Qout==%b,
                mycount=%d\n",counter_checker.Qout,count);
            if(count == 4'd9)
            {
                @0 counter_checker.Qout == 1;
                @(posedge counter_checker.clk);
                count = 0;
            }
            else
            {
                @0 counter_checker.Qout == 0;
                @(posedge counter_checker.clk);
                count = count + 1;
            }
        }
    }
}

```



```

    }
    else
    {
        @(posedge counter_checker.clk);
        count = 0;
        counter_checker.Qout == 0;
    }
    coverage (REPORT,cnt0,"counter_FSM.rpt");
} //end of while (1)
} // end of program counter_test

```

counter_checker.vcon

```

//clock counter.clk period 100
connect input counter_checker.rst_n=counter_tb.U1.rst_n
connect input counter_checker.clk=counter_tb.U1.clk
connect input counter_checker.Qout=counter_tb.U1.Qout
// timescale 100ns/10ns
clock SystemClock period 100

```

counter_checker.vrl

```

counter_checker.vro
counter_coverage.vro

```

counter_checker.proj

```

main counter_checker
    counter_checker.vcon
    counter_checker.vrl

```

counter_coverage.vrh

```

#ifndef INC__TMP_COUNTER_COVERAGE_VRH
#define INC__TMP_COUNTER_COVERAGE_VRH

enum counter_states = S0, S1, S2, S3, S4, S5, S6, S7, S8, S9;

```

```

extern coverage_def counter_sm_cov
(
    bit [3:0] count_state
);
#endif

```

counter_coverage.vr

```

#include <vera_defines.vrh>
#include "counter.if.vrh"

enum counter_states =
    S0 = 4'h0,
    S1 = 4'h1,
    S2 = 4'h2,
    S3 = 4'h3,
    S4 = 4'h4,
    S5 = 4'h5,
    S6 = 4'h6,
    S7 = 4'h7,
    S8 = 4'h8,
    S9 = 4'h9;

coverage_def counter_sm_cov (bit [3:0] count_state)
{
    state s0(S0),s1(S1),s2(S2),s3(S3),s4(S4),s5(S5),s6(S6),s7(S7),
        s8(S8),s9(S9);
    bad_state (not state);

    trans reset(S0->S0);
    trans s0_s1(S0->S1);
    trans s1_s2(S1->S2);
    trans s2_s3(S2->S3);
    trans s3_s4(S3->S4);
    trans s4_s5(S4->S5);
    trans s5_s6(S5->S6);
    trans s6_s7(S6->S7);
    trans s7_s8(S7->S8);
    trans s8_s9(S8->S9);
    trans s9_s0(S9->S0);
    bad_trans (not trans);

    coverage_option = HI;
}

```

RUN

```
#!/usr/bin/csh -f

vera -cmp -ip -h -I. counter_coverage.vr
vera -cmp -ip -I. counter_checker.vr
ls *vro > counter_checker.vrl
echo "main counter_checker" > counter_checker.proj
echo "counter_checker.vcon" >> counter_checker.proj
echo counter_checker.vrl >> counter_checker.proj

vera -proj counter_checker.proj
verilog +vera_pload=counter_checker.proj \
    counter_checker_shell.v ./counter.v
```

1. The first step is to create a new project in the Vera IDE.

14. Vera-to-HDL Task Calls

Vera is an HDL co-simulation. Because of the close interaction between the two simulators in the verification process, it is often useful to call tasks across platforms. For this reason, Vera enables you to call HDL tasks from within Vera and also provides a means to call Vera tasks within HDLs. This chapter discusses how to make these calls and includes these sections:

- Calling HDL Tasks from Vera
- Calling Vera Tasks from Verilog

14.1 Calling HDL Tasks from Vera

Vera allows you to call HDL tasks from Vera and transfer data back and forth between Vera and the HDL. To declare an HDL task in Vera, use the following syntax within your main program module:

```
verilog_task task_name (arguments) "inst_path";
vhdl_task task_name (arguments) "inst_path";
```

task_name - The *task_name* is the name of the HDL task you want to call.

arguments - Task call *arguments* are passed from Vera to the HDL when the task is called.

Arguments can be of the following types: integer and bit field. Note that you can pass strings as bits using this construct: `bit[(8*string_length)-1:0] str`, where *string_length* is the number of characters in string *str*.

inst_path - The *inst_path* is the instantiation path of the task in the HDL. It identifies the HDL task within the HDL hierarchy starting at the top level HDL module.

This is an example of an HDL task call from within Vera:

```
verilog_task chip_init (bit [7:0] init_reg) "top.chip.chip_init";
```

This example calls the Verilog task `chip_init` and passes the bit field variable `init_reg`. The task can be found within the `top.chip` hierarchy in the Verilog declaration.

Because Vera identifies this declaration as a task definition, it must occur in the top code block. However, if you want to call the HDL task from other files, you can define it as an external task using the `extern` construct:

```
extern verilog_task (arguments);
extern vhdl_task (arguments);
```

For example, to declare the `chip_init` task so it can be called from other files, use:

```
extern verilog_task chip_init (bit [7:0] init_reg);
```

14.1.1 HDL Tasks in Vera: Outputs/Inouts

For HDL outputs and inouts, Vera reflects the output value if the Vera formal argument is prefixed by the `var` keyword.

Note – It is important to define the `var` parameter in the HDL as `inout`.

This is an example of the `var` construct with HDL task calls:

```
verilog_task get_status (var bit [7:0] status_word) "top.cpu.status";
verilog_task chip_init (bit [7:0] init_reg);
{
    bit [7:0] word;
    integer i;

    chip_init (8'b0000_0000); // OK
    get_status(word);        // OK
    get_status(i);           // Error: formal is bit[7:0]; actual is integer
    get_status(8'b0000_0000); //Error: formal is var; actual is constant
}
```

The task call of `chip_init` is successful because the Verilog task has been defined to accept any `bit[7:0]` as a parameter. The `get_status(word)` task call is successful because “word” matches the formal variable type specified in the declaration. The `get_status(i)` task call does not succeed because the variable type of the argument (`integer`) does not match the formal declaration (`bit [7:0]`). The `get_status(8'b0000_0000)` task call does not succeed because the argument passed is a constant and the formal declaration is a variable.

14.1.2 Type Checking HDL Tasks in Vera

Vera can check the argument types of HDL task declarations within Vera and any task calls made within Vera. However, Vera cannot access the task definition made within the HDL. So Vera cannot check the argument types in the Vera program against the argument types in the HDL definition. Therefore, you must check that the declarations in Vera are consistent with the HDL task definition.

If the types are not consistent, the results of the task calls are unpredictable. Although the right type of implicit conversions may occur, this is not guaranteed. Instead, it is recommended that you use types which are consistent in Vera and HDL, and do any type conversion separately.

Note – This is equivalent to what happens when calling external functions in C, where the burden of consistency between the extern declarations and the libraries is not on the compiler but on the programmer (or on lint).

14.2 Calling Vera Tasks from Verilog

Vera provides mechanisms to call Vera tasks from the HDL. Currently this is only available with Verilog designs.

14.2.1 Declaring Vera Tasks for Export

If you want to call Vera tasks from an HDL, the tasks must be declared with the **export** construct in the top level Vera file (the **program** file):.

```
export task task_name (arguments) {
    task_contents;
}
```

arguments - The *arguments* are passed from the HDL to Vera when the task is called. They can be of type integer or bit field. Strings can be passed as bits using this construct: `bit[(8*string_length)-1:0] str`, where *string_length* is the number of characters in string *str*. The *arguments* can be **var** and **non-var** variables as well. Default arguments can be specified. However, while they will apply to calls made from Vera code, they will not apply to calls made from the HDL.

task_contents - The *task_contents* can be any Vera code valid in normal task definitions.

Note – You can only export global tasks. You cannot export class methods or functions because of the restriction of function usage within an HDL.

As an example, examine:

```
export task foo(integer num) {
    printf("Task foo() called with num %0d!!\n", num );
}
```

This example declares a task `foo` and passes the argument `num` of type integer.

14.2.2 Calling Exported Vera Tasks

When you declare an export task, a task with the same name is generated in the `vera_shell` module. It is called from the HDL as `vshell.task_name`. For example:

```
vshell.foo(5);
```

However, if the `vera_shell` exists as a co-top-level module (test-top connections between Vera and the HDL signals are done using "HDL_node" constructs in the interface), then the task must be called from the HDL as `vera_shell.task_name`.

Exported tasks behave as normal Vera tasks and can be called within the Vera code.

Vera imposes no restrictions on the contents of exported tasks. Exported tasks can have delays, forks/joins, and calls to other tasks or functions including calls back to HDL tasks.

Warning – Though Vera tasks are re-entrant, Vera task calls from an HDL are NOT re-entrant because of the Verilog task wrapper inside of the `vera_shell`.

14.2.3 Exporting External Tasks

Exported tasks must be declared in the program file. However, exported tasks can be external Vera task declarations. So the definition of the body of the task can be in a separate subroutine file, which is compiled and included separately. If this is the case, only the `export extern` declaration has to be in the program file, and only the following line needs to be in the program file:

```
export extern task task_name(arguments);
```

The Vera task declaration and task definition can be in the same file. If they are both within a program file, you can export the task if either (or both) has the `export` prefix.

In a subroutine file, task definitions cannot be of type `export`. The compiler simply ignores the prefix in this case. Therefore, a task that is declared as an `export extern` in a program but whose body is defined in a separate subroutine file should not have an `export` prefix in the subroutine file. Whether or not a task is exported depends only on how it is declared (or defined) in the program file.

15. Calling C/C++ Functions

This chapter discusses Vera's handling of C and C++ function calls. This chapter includes these sections:

- Declaration and Invocation
- UDF Arguments
- Linking UDFs to Vera
- PLI Support Procedures
- Handling Special Events

15.1 Declaration and Invocation

All Vera UDFs (user-defined functions) must be declared before they are used, as either tasks or functions. The syntax to declare a UDF is:

```
function data_type $udf_name (arguments);  
task $udf_name (arguments);
```

data_type - The *data_type* specifies the data type returned by the function call. Functions can be of type integer, bit, and string. The default data type is integer.

udf_name - The *udf_name* is the name of the UDF being declared.

arguments - The *arguments* can be arguments of type integer, bit and string, and they can include var declarations.

15.2 UDF Arguments

UDF arguments must be handled from the Vera side and the C/C++ side.

15.2.1 Vera UDF Arguments

The size and type of the UDF arguments in the function or task call must match exactly those in the declaration. If they do not, a compilation error occurs.

For example, this task declaration and call result in a compilation error:

```
task $my_task (integer i);  
$my_task(string_a);
```

Vera also supports bidirectional arguments with UDFs using the `var` construct. Arguments passed using the `var` construct can both pass and receive data when the function or task is called. For example:

```
task $my_task (integer i, var bit [7:0] mydata);
```

When using bidirectional arguments, you must be careful to type check the call with the declaration. For instance, given the above task declaration, this call results in a compilation error:

```
$my_task(100,200);
```

This call results in an error because a constant is passed (200) where a variable is expected (*mydata*).

You can specify an unknown number of arguments by using an asterisk (*) in place of actual arguments in the function or task call. For example:

```
task $my_printf(*);
```

If you use this construct, the arguments must be of type integer or bit.

15.2.2 C/C++ UDF Arguments

Integers and strings are passed as is. They do not need a Vera data structure to be passed to Vera.

The bit vector data structure that Vera uses is:

```
typedef struct t_vecval
{
    int avalbits;
    int bvalbits;
} s_vecval, *p_vecval
```

The values that this data structure yields depend on the *aval* and *bval* integer values. Table 15-1 shows the values.

Table 15-1 Bit vector data structure values

aval	bval	Value
0	0	0
1	0	1
0	1	Z
1	1	X

The Vera Data data structure is used to transfer values between Vera and the UDF. The Vera Data data structure is:

```
typedef struct VERA_DATA
{
    int type;
    union
```

```

{
    int int_value;
    char string;
    struct
    {
        int expr_ngroups;
        int expr_vec_size;
        s_vecval expr_value_p;
    } bit_val;
} data;
} vera_data;

```

type - The *type* must be **VERA_INT_X**, **VERA_INT**, **VERA_BIT**, or **VERA_STRING**, depending on the value type being passed. The integer equivalents of these types are 1, 2, 3, and 4 respectively.

int_value - The *int_value* is the integer value of any data being passed.

string - The *string* is the string value of any data being passed. It can be NULL if the string on the Vera side is NULL.

expr_ngroups - The *expr_ngroups* is the number of groups in the expression value.

expr_vec_size - The *expr_vec_size* is the number of bits in the expression value.

expr_value_p - The *expr_value_p* is a pointer to the expression value.

More information concerning this data structure can be found in the *Verilog PLI Reference Manual*.

15.3 Linking UDFs to Vera

To link UDFs to Vera, you must modify a UDF table in the *vera_user.c* file and create object files.

15.3.1 Vera UDF Table

To link UDFs to Vera, you must put an entry into the Vera UDF Table for each user routine. The table *\$VERA_HOME/lib/vera_user.c*. The syntax for a table entry is:

```

vera_user_call_entry Vera_UDF_Table[] = {
    type, "proc_name", proc, misc_proc, data, "comment"
}

```

type - The *type* should be either **VERA_USER_TASK** or **VERA_USER_FUNC**.

proc_name - The *proc_name* is the name of the UDF being linked, as it is known to the user.

proc - The *proc* is the actual UDF name, as it is called within Vera. It is called with the argument *data*.

misc_proc - The *misc_proc* is called when Vera starts and when it transfers control back and forth with the HDL. This procedure is discussed in Section 15.5, "Handling Special Events."

data - The argument *data* is passed to the UDF call. It must be an integer.

comment - The *comment* can be any user comment.

The Vera UDF Table is loaded dynamically at runtime from the library *vera_local.dl*.

15.3.2 Object Files and *vera_local.dl* for Verilog

The *vera_local.dl* file is a dynamic library file that you must manually generate when you create your UDF object files. There are two steps you must take to create the *vera_local.dl* file:

1. Compile *vera_user.c* and all your library files with the appropriate position independent code options to create the *vera_local.dl* file.
2. Link the *vera_local.dl* file with the necessary object files and libraries.

This process is platform-specific.

HPUX

To create the *vera_local.dl* file with object files:

```
cc -c +z -I$VERA_HOME/lib c_files vera_user.c
```

OR

```
gcc -fpic -I$VERA_HOME/lib c_files vera_user.c
```

c_files - The *c_files* are the UDF files you are linking with Vera.

After compiling *vera_user.c*, link the *vera_local.dl* file with the necessary object files:

```
ld -b +e syssci_prod_entry + errno -o vera_local.dl o_files \
vera_user.o vera_kernel -lc
```

o_files - The *o_files* are the compiled UDF object files.

Solaris

To create the *vera_local.dl* file with object files:

```
cc -K pic -c -I$VERA_HOME/lib c_files vera_user.c
```

OR

```
gcc -fpic -c -I. -I$VERA_HOME/lib c_files vera_user.c
```

c_files - The *c_files* are the UDF files you are linking with Vera.

After compiling *vera_user.c*, link the *vera_local.dl* file with the necessary object files:

```
ld -G -Bsymbolic -o vera_local.dl o_files vera_user.o vera_kernel \
-lsocket -lnsl -lintl -lc
```

o_files - The *o_files* are the compiled UDF object files.

SunOS

To create the *vera_local.dl* file with object files:

```
cc -c -I$VERA_HOME/lib c_files vera_user.c
```

c_files - The *c_files* are the UDF files you are linking with Vera.

After compiling *vera_user.c*, link the *vera_local.dl* file with the necessary object files:

```
ld -o vera_local.dl o_files vera_user.o vera_kernel
```

o_files - The *o_files* are the compiled UDF object files.

15.3.3 Object Files and vera_mti.dl for VHDL

The *vera_mti.dl* file is a dynamic library file that you must manually generate when you create your UDF object files. There are two steps you must take to create the *vera_local.dl* file:

1. Compile *vera_user.c* and all your library files with the appropriate position independent code options to create the *vera_local.dl* file.
2. Link the *vera_local.dl* file with the necessary object files and libraries.

This process is platform-specific.

HPUX

To create the *vera_mti.dl* file with object files:

```
cc -c +z -I$VERA_HOME/lib c_files vera_user.c
```

OR

```
gcc -fpic -I$VERA_HOME/lib c_files vera_user.c
```

c_files - The *c_files* are the UDF files you are linking with Vera.

After compiling *vera_user.c*, link the *vera_mti.dl* file with the necessary object files:

```
ld -b +e syssci_prod_entry + errno -o vera_mti.dl o_files \
vera_user.o vera_kernel_mti -lc
```

o_files - The *o_files* are the compiled UDF object files.

Solaris

To create the *vera_mti.dl* file with object files:

```
cc -K pic -c -I$VERA_HOME/lib c_files vera_user.c
```

OR

```
gcc -fpic -c -I. -I$VERA_HOME/lib c_files vera_user.c
```

c_files - The *c_files* are the UDF files you are linking with Vera.

After compiling *vera_user.c*, link the *vera_mti.dl* file with the necessary object files:

```
ld -G -Bsymbolic -o vera_mti.dl o_files vera_user.o vera_kernel_mti \
-lsocket -lnsl -lintl -lc
```

o_files - The *o_files* are the compiled UDF object files.

SunOS

To create the *vera_mti.dl* file with object files:

```
cc -c -I$VERA_HOME/lib c_files vera_user.c
```

c_files - The *c_files* are the UDF files you are linking with Vera.

After compiling *vera_user.c*, link the *vera_mti.dl* file with the necessary object files:

```
ld -o vera_mti.dl o_files vera_user.o vera_kernel_mti
```

o_files - The *o_files* are the compiled UDF object files.

15.4 PLI Support Procedures

Vera provides several procedures designed for PLI support.

vera_num_arg()

The *vera_num_arg()* system function returns the number of arguments in the UDF that it is called in.

The syntax is:

```
vera_num_arg();
```

vera_get_arg()

The *vera_get_arg()* system function returns a Vera data structure with the values in the specified argument. The syntax is:

```
vera_get_arg(index);
```

index - The *index* is an integer that specifies which data structure to return.

The function returns the data structure associated with the specified argument. The values returned should be used without any modifications. You can free it manually or by calling *vera_free_arg()*.

vera_free_arg()

The `vera_free_arg()` system function frees a Vera data structure returned by `vera_get_arg()`. The syntax is:

```
vera_free_arg(vera_data);
```

vera_data - *Vera_data* is a pointer to the data structure you want to free.

When `vera_free_arg()` is called, it frees all memory allocated by `vera_get_arg()`, including bit and string data.

vera_return_value()

The `vera_return_value()` system function sets the return value for the UDF. The syntax is:

```
vera_return_value(vera_data);
```

vera_data - *Vera_data* is a pointer to the data structure you want to return a value from.

The `vera_return_value()` system function returns the value associated with the specified data structure. It may be called only one time before returning.

Note – Calling `vera_return_value()` multiple times before returning may cause a crash.

vera_put_arg()

The `vera_put_arg()` system function sets the return value for a var argument. The syntax is:

```
vera_put_arg(index, vera_data);
```

int - The *index* specifies which argument is assigned a value.

vera_data - *Vera_data* is a pointer to the data structure containing the value to be assigned.

When the `vera_put_arg()` system function is called, it assigns the value associated with the given data structure to the specified argument.

15.5 Handling Special Events

When an HDL and Vera start or reset, your C functions may need to perform certain operations before they are ready to be used. Also, they may need to know when Vera and the HDL are passing control back and forth. To handle these events, you can write miscellaneous functions, which you call in the Vera UDF Table (see Section 15.3.1, "Vera UDF Table").

When one of the predefined reasons occurs, the miscellaneous routine is called. Along with the *data* parameter, it transfers a parameter to the function call that is based on why the routine was called. Table 15-2 lists the *VERA_REASONS* and their values.

Table 15-2 VERA_REASON Values

VERA_REASON	Value
VERA_REASON_INIT	1
VERA_REASON_RESET	2
VERA_REASON_RESUME	3
VERA_REASON_SUSPEND	4
VERA_REASON_EXIT	5

This function can be used to handle special events or set conditions necessary before UDF calls can be made.

Example 9: User-Defined Functions

This is an example of UDF usage. This example covers the following constructs:

- UDF code
- UDF call from Vera
- RUN script

This example includes these files:

- *udfcode.c*
- *test.vr*
- *RUN* (a run script)

udfcode.c

```
#include <stdio.h>
#include "vera_user.h"

/* task $upctask(integer i, string str1, var string str2); */
int upctask(int data)
{
    int nargs = vera_num_arg(), i;
    vera_data *arg1, *arg2, *arg3;
    vera_data retval;
    char *procn = "upctask", *str1, *str2;

    if (nargs != 3)
    {
        fprintf(stderr, "upctask: Wrong number of args (%d)\n", nargs);
        exit(1);
    }
    arg1 = vera_get_arg(0);
    arg2 = vera_get_arg(1);
    arg3 = vera_get_arg(2);
    if (arg1->type != VERA_INT)
    {
        fprintf(stderr, "upctask: 1st arg is not an integer\n");
        exit(1);
    }
    if (arg2->type != VERA_STRING)
    {
        fprintf(stderr, "upctask: 2nd arg is not a string\n");
        exit(1);
    }
    if (arg3->type != VERA_STRING)
    {
        fprintf(stderr, "upctask: 3rd arg is not a string\n");
        exit(1);
    }
    /* ... (rest of the function code) ... */
}
```

```

        fprintf(stderr, "upctask: 1st arg is not a string\n");
        exit(1);
    }
    if (arg3->type != VERA_STRING)
    {
        fprintf(stderr, "upctask: 1st arg is not a string\n");
        exit(1);
    }
    i = arg1->data.int_val;
    str1 = arg2->data.string;
    str2 = arg3->data.string;
    fprintf(stderr, "task %s: i=%d str1='%s' str2='%s'\n", procn, i,
            str1, str2);
    retval.type = VERA_STRING;
    retval.data.string = "arg task, task";
    vera_put_arg(2, &retval);
    vera_free_arg(arg1);
    vera_free_arg(arg2);
    vera_free_arg(arg3)
}

/* function string $upcfunc(integer i, string str1, var string str2); */
int upcfunc(int data)
{
    int nargs = vera_num_arg(), i;
    vera_data *arg1, *arg2, *arg3;
    vera_data retval;
    char *procn = "upcfunc", *str1, *str2;

    if (nargs != 3)
    {
        fprintf(stderr, "upcfunc: Wrong number of args (%d)\n", nargs);
        exit(1);
    }
    arg1 = vera_get_arg(0);
    arg2 = vera_get_arg(1);
    arg3 = vera_get_arg(2);
    if (arg1->type != VERA_INT)
    {
        fprintf(stderr, "upcfunc: 1st arg is not an integer\n");
        exit(1);
    }
    if (arg2->type != VERA_STRING)
    {
        fprintf(stderr, "upcfunc: 1st arg is not a string\n");

```

```

        exit(1);
    }
    if (arg3->type != VERA_STRING)
    {
        fprintf(stderr,"upcfunc: 1st arg is not a string\n");
        exit(1);
    }
    i = arg1->data.int_val;
    str1 = arg2->data.string;
    str2 = arg3->data.string;
    fprintf(stderr,"func %s: i=%d str1='%s' str2='%s'\n", procn, i, str1,
        str2);
    retval.type = VERA_STRING;
    retval.data.string = "arg func, func";
    vera_put_arg(2,&retval);
    retval.data.string = "ret func, func";
    vera_return_value(&retval);
    vera_free_arg(arg1);
    vera_free_arg(arg2);
    vera_free_arg(arg3);
}

/* misc procedure calls */

int upcmisc1(int reason, int data)
{
    printf("upcmisc1 called with data=%d and reason=",data);
    switch (reason)
    {
        case VERA_REASON_INIT:printf("VERA_REASON_INIT\n"); break;
        case VERA_REASON_RESET:printf("VERA_REASON_RESET\n"); break;
        case VERA_REASON_RESUME:printf("VERA_REASON_RESUME\n"); break;
        case VERA_REASON_SUSPEND:printf("VERA_REASON_SUSPEND\n"); break;
        case VERA_REASON_EXIT:printf("VERA_REASON_EXIT\n"); break;
        default: printf("????????????????\n"); break;
    }
}

int upcmisc2(int reason, int data)
{
    printf("upcmisc2 called with data=%d and reason=",data);
    switch (reason)
    {
        case VERA_REASON_INIT:printf("VERA_REASON_INIT\n"); break;
        case VERA_REASON_RESET:printf("VERA_REASON_RESET\n"); break;
        case VERA_REASON_RESUME:printf("VERA_REASON_RESUME\n"); break;

```

```

        case VERA_REASON_SUSPEND:printf("VERA_REASON_SUSPEND\n"); break;
        case VERA_REASON_EXIT:printf("VERA_REASON_EXIT\n"); break;
        default: printf("????????????????\n"); break;
    }
}

```

test.vr

```

#include <vera_defines.vrh>

task $upctask(integer i, string str1, var string str2);
function string $upcfunc(integer i, string str1, var string str2);

program test
{
    string a, b, c;
    a = "I am a";
    b = "I am b";
    c = "I am c";
    $upctask(1,"I am here",c);
    printf("a='%s' b='%s' c='%s'\n",a,b,c);
    c = $upcfunc(1,a,b);
    printf("a='%s' b='%s' c='%s'\n",a,b,c);
}

```

RUN

```

#!/bin/csh -f
#!/bin/csh -fxv

set ARCH = `/usr/local/bin/ncpu`
echo $VERA_HOME

rm -rf vera_user.h vera_user.c vera_kernel libVERA.a vera_cs_local \
    vera_local.dl vera_user.o udfcode.o test.vro simv simv.daidir \
    verilog.log >& /dev/null

cp $VERA_HOME/lib/{vera_user.h,vera_user.c,vera_kernel,libVERA.a} .
sed -e '/\/\* user entries go here \*\/s/^.*$/#include "udf.c"/\' \
    vera_user.c > temp

rm -f vera_user.c
cat - temp > vera_user.c <<END

#include <stdio.h>
#include "udf.h"

END

```

```

rm -f temp vera_local.dl
set csrc = (vera_user.c udfcode.c)
set cos  = (vera_user.o udfcode.o)
setenv SSI_LIB_FILES ./vera_local.dl
vera -cmp test.vr
if ($status) exit 1
switch ($ARCH)
case "SOL":
    cc -DNO_VERILOG_SIM -K pic -c $csrc |& egrep -v '^vera_user.c:$' \
        ^udfcode.c:$'
    ld -G -Bsymbolic -o vera_local.dl $cos vera_kernel -lsocket -lnsl \
        -lintl -lc
    cc -DNO_VERILOG_SIM -o vera_cs_local $csrc libVERA.a -lsocket -lnsl \
        -lintl -lm |& egrep -v '^vera_user.c:$|^udfcode.c:$'
    vcs -q test.vshell -l vcs.log -P $VERA_HOME/lib/vera_pli.tab \
        $VERA_HOME/lib/libSysSciTaskpic.a >& /dev/null
    if ($status) exit 1
    breaksw
case "SUN4":
    gcc -DNO_VERILOG_SIM -fpic -c $csrc
    ld -o vera_local.dl $cos vera_kernel
    chmod +w libVERA.a
    ranlib libVERA.a
    gcc -DNO_VERILOG_SIM -o vera_cs_local $csrc libVERA.a -lm
    vcs -q test.vshell -l vcs.log -P \
        $VERA_HOME/lib/vera_pli.tab $VERA_HOME/lib/libSysSciTask.a \
        >& /dev/null
    if ($status) exit 1
    breaksw
case "HPUX-10":
    gcc -DNO_VERILOG_SIM -fpic -c $csrc
    ld -b +e syssci_prod_entry +e errno -o vera_local.dl \
        $cos vera_kernel -lc -lm
    gcc -DNO_VERILOG_SIM -o vera_cs_local $csrc libVERA.a -lm
    vcs -q test.vshell -l vcs.log -Mupdate=1 -gen_c -P \
        $VERA_HOME/lib/vera_pli.tab $VERA_HOME/lib/libSysSciTaskpic.a \
        -LDFLAGS "-E -a archive_shared" -ldld >& /dev/null
    if ($status) exit 1
    breaksw
endsw

```

```
vera -cmp test.vr
if ($status) exit 1
./vera_cs_local test.vro
verilog test.vshell -l verilog.log +vera_load=test.vro
./simv +vera_load=test.vro

rm -rf vera_user.h vera_user.c vera_kernel libVERA.a vera_cs_local\
vera_local.dl vera_user.o udfcode.o test.vro simv simv.daidir >& \
/dev/null
```

2000-2001 Synopsys, Inc.

16. VERA-SV API

This chapter introduces Vera-SV API, which is used for hardware/software co-verification and distributed simulation. It includes these sections:

- Vera-SV Motivations
- Vera-SV Overview
- The Vera Side
- The C/C++ Side
- Troubleshooting
- Backward Compatibility

16.1 Vera-SV Motivations

The key motivations behind the development of Vera-SV are hardware/software co-verification, distributed simulations, and a master-slave configuration where the C side controls the simulation.

16.1.1 Hardware/Software Co-Verification

HW/SW co-verification involves the concurrent development of software and hardware. In this methodology, software components (drivers, debuggers, application programs) are compiled to run at full speed on a workstation and interact in a bus-cycle accurate fashion with a simulated hardware model.

This API supports HW/SW co-verification by enabling C/C++ applications (e.g. drivers, debuggers, application programs) and Vera programs to execute remote calls to each other. C/C++ applications are compiled into native binary executables and run on the host workstation at full speed. They use the API to interact with hardware models (Verilog, VHDL, C) running under the control of a Vera program.

With the API, C/C++ applications call tasks and functions in the Vera program which then probe, stimulate, and monitor the hardware model. In a similar fashion, the Vera program can react to an event generated by the hardware model and call the C/C++ application to notify it of an asynchronous event (like an interrupt, for instance).

16.1.2 Distributed Simulation

Distributed simulation is often used with, but not limited to, HW/SW co-verification. Distributed simulation involves verifying large designs by partitioning verification tasks to run concurrently across multiple processors. This API supports distributed simulation by enabling multiple Vera programs and C/C++ applications to execute remote calls to each other across a network of workstations. For example, an application program on one host can be executing assembly code for a particular processor and interacting via the API with Vera programs on another host that simulate the board containing the processor.

Note – Both UNIX and NT workstations are supported.

16.1.3 Master-Slave Configuration

The traditional and default setup for Vera/C interfaces has Vera as the master. In this setup, Vera controls the simulation time and determines how the calls to C are made. However, sometimes it is useful to have the C side control the simulation completely and dictate the simulation time along with all simulation activity. Vera-SV provides the flexibility for such a setup.

16.2 Vera-SV Overview

The Vera System Verifier API is a collection of predefined Vera functions and a C programmer's library. Together, they enable the development of hardware and software co-verification and distributed simulation environments.

With this API, verification engineers can build interfaces between system components modeled with Vera and C/C++. This API enables Vera programs to communicate and synchronize with other Vera programs and with independent C/C++ applications via remote calls.

The API's remote call technology permits a natural way to model system components:

- Remote calls initiated from Vera programs are inherently multi-threaded (Vera threads, not UNIX/NT threads). This allows your hardware models to mimic the concurrency of real hardware. For example, a Vera program can have multiple tasks waiting concurrently for their remote calls to execute. While these suspended tasks are waiting, other Vera tasks will continue their execution.
- Remote calls from a client to a server may be either blocking or non-blocking. Blocking calls have an implicit handshake; they wait for the server to execute the call before returning. Non-blocking calls are equivalent to message passing; they send the call to the server and return immediately (possibly before the server has a chance to execute the call).
- The remote call mechanism efficiently passes large chunks of data between simulations.

The API creates point-to-point (1-to-1) connections. A connection consists of a server/client pair; the server is on one side of the connection; the client is on the other. A single process may create both client and server connections, but it must not connect to itself.

A single process may also create multiple server and client connections, but each server connects to exactly one client, and each client connects to exactly one server.

Using the API is straight-forward and consists of these three basic steps:

1. Define a network of server/client, i.e., components which will talk to each other.
2. Bring up the servers/clients network.
3. Execute the remote calls.

16.3 The Vera Side

This section describes how to make remote calls from Vera programs to other Vera programs. All connections between the Vera programs need to be defined and brought up before remote calls are made. The Vera functions that define the network of connections are `vsv_make_server()` and `vsv_make_client()`.

16.3.1 Initializing Connections

Vera has a set of functions used to initialize connections.

`vsv_make_server()` and `vsv_make_client()`

Vera provides the system functions `vsv_make_server()` and `vsv_make_client()` to initialize connections. The syntax is:

```
vsv_make_server(port, authentication [, verbose]);
vsv_make_client("host", port, authentication);
```

port - The *port* indicates a virtual port number used for a single client/server pair during a session. It can be any positive integer.

authentication - The *authentication* specifies an authentication code to use while bringing up connections. If the server specifies a zero value, no authentication is needed for the connection, and the value provided by the client is discarded. If the server authentication is not zero, the client and server authentications must match.

verbose - The *verbose* modifier can be omitted or `VERBOSE`. If it is set, the server outputs messages if errors occur while processing remote client submissions.

host - The *host* is a string that specifies the IP address or host name of the server.

Both these functions return a non-zero integer identifier that is used for the subsequent use of the connection. A returned zero-value means the function failed and the connection has not been set up.

Each call to `vsv_make_server()` must be matched by one and only one call to `vsv_make_client()` in another Vera simulation.

This is an example of the `vsv_make_server()` and `make_host()` system functions:

```
integer servconn;
servconn = vsv_make_server(5555, 0, VERBOSE);

integer client1, client2;
client1 = vsv_make_client ("localhost", 5555, 0);
client2 = vsv_make_client ("171.64.195.82", 7777, 0);
```

This example creates two clients for the host. The authentications for each client are 0, which matches the server authentication. Each client/server pair is opened on a separate port.

`vsv_up_connections()`

Once all connections have been configured, the `vsv_up_connections()` system function is used to make all of them active. The syntax is:

```
vsv_up_connections(timeout);
```

timeout - The *timeout* specifies the time limit allowed to bring up connections. It is specified in seconds per connection.

The function returns 0 if all connections have been activated or -1 otherwise. The `vsv_up_connections()` system function can only be called once in each simulation. If it fails, none of the connections made by `vsv_make_client()` or `vsv_make_server()` are usable, and you need to re-start the simulations if you want to use the Vera-SV API.

After a successful activation, the connections are ready to process and/or submit remote calls.

This is an example of the `vsv_up_connections()` system function:

```
vsv_up_connections(30);
```

In this example, if the Vera program has previously configured a server and a client connection, `vsv_up_connections(30)` times out after 60 seconds.

`vsv_close_conn()`

Connections can be closed with the `vsv_close_conn()` system function. The syntax is:

```
vsv_close_conn(connection);
```

connection - The *connection* specifies which connection to close. It is the integer identifier passed by `vsv_make_server()` when the connection is created.

This function return -1 if an error occurs, otherwise returns zero.

This is an example of the `vsv_close_conn()` system function:

```
myconn=vsv_make_client("localhost", 5555, 0);
vsv_close_conn (myconn);
```

16.3.2 Servicing Remote Calls

When a Vera program is a server, all global Vera task and functions are available as remote procedures to the client, and servicing remote calls is transparent. At the end of every cycle, the Vera simulation engine will process arriving remote calls. The remote calls will then be dispatched to the tasks and functions in the Vera program. When these tasks and functions complete, their returned values and VAR parameters are automatically sent back to the client.

Remote calls run simultaneously with the main Vera application. It is up to the user to synchronize the remote calls with the main execution if necessary.

16.3.3 Submitting Remote Calls

These are the two system functions which can submit calls from a Vera program to a remote application: `vsv_call_func()` and `vsv_call_task()`. The syntax to call them is:

```
vsv_call_func(connection, time_mode, func_name, return_value [, opt_args]);
vsv_call_task(connection, time_mode, task_name [, opt_args]);
```

connection - The *connection* is the connection identifier returned by `vsv_make_client()`.

time_mode - The *time_mode* must be `WAIT`, which suspends simulation time until the call is completed, or `NO_WAIT`, which allows the simulation time to advance while the call is in execution.

func_name/task_name - The *func_name* or *task_name* is a string with the name of the remote procedure to be executed.

return_value - The *return_value* must be a variable (not an expression) that matches the type returned by the remote function. The *return_value* can be of type integer, bit vector, or string. The value of the variable is set by the remote function call.

opt_args - The *opt_args* are optional arguments that must match the corresponding arguments on the function or task side. The *opt_args* can be of type integer, bit vector, or string. They can be variables if the remote argument specifies a `var` argument, or expressions if the remote argument is not of type `var`.

These are examples that illustrate valid uses of arguments:

Remote prototype

Valid local calls

```

integer result, c, i[10];
string mystr;
bit [5:0] r;
INTEGER factorial (NONVAR/INTEGER)    vsv_call_func(...factorial,result,5)
                                       vsv_call_func(factorial,result,i[5])
                                       vsv_call_func(...factorial,result,c)

printValue ( NONVAR/STRING NONVAR/BIT ) vsv_call_task(...printValue,"foo",2'bx1)
                                       vsv_call_task(...printValue,mystr,r)

myfoo ( VAR/STRING VAR/INTEGER )       vsv_call_task(...myfoo,mystr,c)

```

The `vsv_call_func()` and `vsv_call_task()` system functions suspend the calling Vera thread until the call has completed. Other Vera threads that are not suspended can continue to run, as long as they do not advance simulation time. Both `vsv_call_func()` and `vsv_call_task()` return a value of 0 on successful completion and a non-zero value otherwise.

This is an example of the `vsv_call_func()` and `vsv_call_task()` system functions:

```

integer result;
integer myconn;
bit [7:0] b = 8'bxxx000zz;

myconn = vsv_make_client("localhost", 3333, 0, VERBOSE);
vsv_up_connections();
if (vsv_call_func(myconn, NO_WAIT, "factorial", result, 5) != 0)
    printf("Error in factorial call: %s\n",vsv_get_conn_err());
if (vsv_call_task(myconn, WAIT, "printValue", "BitValue", b) != 0)
    printf("Error in printValue call: %s\n",vsv_get_conn_err());

```

16.3.4 Managing errors

When any of the API functions return an error value, the `vsv_get_conn_err()` function can be called to obtain a message describing the cause of the failure. The syntax is:

```
vsv_get_conn_err();
```

The `vsv_get_conn_err()` returns a message describing the cause of the failure. Possible reasons of failure are a: dropped connection, mismatched authentication, mismatched task/function prototype, etc.

This is an example of the `vsv_get_conn_err()` system function:

```

if (vsv_call_task(conn, NO_WAIT, "myfoo", mystr, c) != 0)
    printf("Error in myfoo call: %s\n",vsv_get_conn_err());

```

16.3.5 Master-Slave Configuration

Vera-SV provides the `vsv_wait_for_input()` and `vsv_wait_for_done()` system tasks to facilitate the use of Vera in a Master-Slave environment where Vera is the slave (server) and a C/C++ process is the master (client).

`vsv_wait_for_input()`

The `vsv_wait_for_input()` system task suspends the current Vera thread until a remote client function request is received. The syntax is:

```
vsv_wait_for_input(wait_mode);
```

wait_mode - The *wait_mode* can be `WAIT`, which suspends both the Vera thread and the simulation time until a remote client function request is received, or `NO_WAIT`, which allows other Vera threads to increment the simulation time as necessary.

When the `vsv_wait_for_input()` system task is called, the current Vera thread is suspended. If the wait mode is set to `WAIT`, the simulation time is also suspended. All Vera threads waiting for the simulation time to advance are suspended as well. However, if the wait mode is set to `NO_WAIT`, other concurrent Vera threads continue to execute and advance simulation time. As soon as a remote client function request is received, the thread resumes execution. If there are already remote calls being processed at the time the call is made, the task does not suspend the Vera thread.

`vsv_wait_for_done()`

The `vsv_wait_for_done()` system task suspends the current thread until all of the remote client calls have been completely processed and the replies returned. The syntax is:

```
vsv_wait_for_done();
```

As soon as replies have been returned to the client, the thread resumes execution.

Note - The remote client call is considered to have completed when a reply is returned, even if background Vera threads that were created by the remote call are still active.

This is an example of the `vsv_wait_for_input()` and `vsv_wait_for_done()` system tasks:

```
repeat (3)
{
    vsv_wait_for_input(WAIT);
    vsv_wait_for_done();
}
```

This example suspends the current thread and the simulation time until a remote call is received. The requested function is executed, and simulation time is advanced during that execution as necessary. Then the thread suspends until the next call is made. This cycle is repeated three times.

16.3.6 Separate Client and Server Example

This example demonstrates how one Vera program calls a function in another Vera program. A program running on a host named "rigel" calls the "multiply" function in a program running on "ceti".

Program Running on Host "ceti"

```
#include <vera_defines.vrh>

program ceti
{
    event rigel_called_me;
    integer connection;

    // the program running on rigel will call this
    // function integer multiply(integer a, integer b)
    {
        multiply = a * b;
        trigger(ONE_BLAST, rigel_called_me);
    }

    // make this program a server
    connection = vsv_make_server(2222,0,VERBOSE);
    if (connection == 0)
        error("vsv_make_server failure\n");
    if (vsv_up_connections(30))
        error("vsv_up_connections failure\n");

    // wait for rigel to call me
    sync(ALL, rigel_called_me);
    printf("ceti is done\n");
    vsv_close_conn(connection);
}
```

Program Running on Host "rigel"

```
#include <vera_defines.vrh>

program rigel
{
    integer connection;
    integer product;
```

```

// make this program a client
connection = vsv_make_client("ceti",2222,0);
if (connection == 0)
    error("vsv_make_client failure\n");
if (vsv_up_connections(30))
    error("vsv_up_connections failure\n");

// have remote host multiply 3 x 7
if (vsv_call_func(connection, WAIT, "multiply", product, 3, 7))
    error("vsv_call_func failure\n");
printf("3 * 7 = %d\n", product);
vsv_close_conn(connection);
}

```

16.3.7 Combination Client-Servers Example

This example demonstrates how two Vera programs can call each other. Each Vera program operates as both a client and a server.

One program runs on a host named "ceti"; one on a host named "rigel". Both programs initialize a local semaphore to be empty and fork a thread to wait on that semaphore. Each program then calls a remote task in the other program to unlock the semaphore for the other program.

Program Running on "ceti"

```

#include <vera_defines.vrh>

program ceti
{
    integer server, client;
    integer semaphore_id;

    // the program running on rigel will call this task
    task ceti_unlock(integer semaphore_value)
    {
        printf("rigel unlocked me\n");
        semaphore_put(semaphore_id, semaphore_value);
    }

    // initialize my semaphore to empty
    semaphore_id = alloc(SEMAPHORE, 0, 1, 0);
    server = vsv_make_server(2222,0,VERBOSE);
    if (server == 0)
        error("vsv_make_server failure\n");
    client = vsv_make_client("rigel",3333,0);
    if (client == 0)
        error("vsv_make_client failure\n");
}

```

```

    if (vsv_up_connections(30))
        error("vsv_up_connections failure\n");
    fork
    {
        // unlock rigel
        if (vsv_call_task(client, NO_WAIT, "rigel_unlock", 1) != 0)
            error("unlock on rigel failure\n");
    }
    {
        // wait for rigel to unlock me
        semaphore_get(WAIT, semaphore_id, 1);
    }
    join
    printf("ceti is done\n");
    vsv_close_conn(server);
    vsv_close_conn(client);
}

```

Program Running on "rigel"

```

#include <vera_defines.vrh>

program rigel
{
    integer server, client;
    integer semaphore_id;

    // the program running on ceti will call this task
    task rigel_unlock(integer semaphore_value)
    {
        printf("ceti unlocked me\n");
        semaphore_put(semaphore_id, semaphore_value);
    }

    // initialize my semaphore to empty
    semaphore_id = alloc(SEMAPHORE, 0, 1, 0);
    server = vsv_make_server(3333,0,VERBOSE);
    if (server == 0)
        error("vsv_make_server failure\n");
    client = vsv_make_client("ceti",2222,0);
    if (client == 0)
        error("vsv_make_client failure\n");
    if (vsv_up_connections(30))
        error("vsv_up_connections failure\n");
    fork

```



```

{
    // unlock ceti
    if (vsv_call_task(client, NO_WAIT, "ceti_unlock", 1) != 0)
        error("unlock on ceti failure\n");
}
{
    // wait for ceti to unlock me
    semaphore_get(WAIT, semaphore_id, 1);
}
join
printf("rigel is done\n");
vsv_close_conn(server);
vsv_close_conn(client);
}

```

16.4 The C/C++ Side

This section describes how to make remote calls between C/C++ applications and Vera programs. All connections between the C/C++ applications and Vera programs need to be defined and brought up before remote calls are made.

The API procedures for C/C++ applications are similar to their Vera counterparts, and only their prototypes are described in this document.

For a complete description of the API procedures please see the "\$VERA_HOME/lib/vsv.h" file.

These are the prototypes:

```

extern void *vsv_MakeServer (
    int port,
    int authentication
);
extern void *vsv_MakeClient (
    char *host,
    int port,
    int authentication
);
extern int vsv_UpConnections (
    int timeout
);
extern int vsv_CallFunc (
    void *connection,
    char *name,
    vsvArgT *returnVal,
    int numberArgs,
    vsvArgT *args
);
extern int vsv_CallTask (
    void *connection,
    char *name,

```

```

vsvCallWaitingT    wait,
int                numberArgs,
vsvArgT            *args
);
extern int vsv_CloseConn (
    void *connection
);
extern char *vsv_GetErrorMesg (void);

```

In addition to the above, the API also provides procedures for C/C++ applications that will function as servers. C/C++ servers need to pre-register functions and tasks that will be made available to clients by calling `vsv_RegisterFunc` and `vsv_RegisterTask` API procedures. Also, C/C++ servers must explicitly call `vsv_AnswerCalls` to service incoming function and task calls from the clients.

Note – This is not true if the application is both a server and a client, in which case incoming calls will be automatically serviced if the application is waiting for a `vsv_CallFunc` or `vsv_CallTask` to complete. Here are the API prototypes for servers:

```

extern int vsv_RegisterFunc (
    void                *connection,
    char                *name,
    vsvArgT              *returnVal,
    int                 numberArgs,
    vsvArgT              *args,
    vsvFuncCbT           fcb
);
extern int vsv_RegisterTask (
    void                *connection,
    char                *name,
    int                 numberArgs,
    vsvArgT              *args,
    vsvTaskCbT           tcb
);
extern int vsv_AnswerCalls (
    void                *connection,
    vsvCallHandlingT     how
);

```

16.4.1 . Calling a Vera Task from a C-program Example

This example demonstrates how a C-program running on host named "ceti" calls a task in a Vera program running on a host named "rigel".

C Program Running on "ceti"

```
#include <vsv.h>
```

```

main()
{
    vsvArgT arg;
    void *connection;

    /* make this program a client of rigel */
    connection = vsv_MakeClient("rigel",2222,0);
    if(connection == NULL)
        perror(vsv_GetErrorMesg());
    if(vsv_UpConnections(30))
        perror(vsv_GetErrorMesg());

    /* call the remote task passing an integer argument set to 1 */
    arg.vtype = VSV_NONVAR_ARG;
    arg.dtype = VSV_INTEGER_ARG;
    arg.data.integerVal = 1;
    if(vsv_CallTask(connection, "unlock", VSV_WAIT, 1, &arg))
        perror(vsv_GetErrorMesg());
    (void)printf("ceti is done\n");
    (void)vsv_CloseConn(connection);
    exit(0);
}

```

Vera Program Running on "rigel"

```

#include <vera_defines.vrh>

program rigel
{
    integer connection;
    integer semaphore_id;

    // the program running on ceti will call this task
    task unlock(integer semaphore_value)
    {
        printf("ceti unlocked me\n");
        semaphore_put(semaphore_id, semaphore_value);
    }

    // initialize my semaphore to empty
    semaphore_id = alloc(SEMAPHORE, 0, 1, 0);

    // make this program a server
    connection = vsv_make_server(2222,0,VERBOSE);
    if (connection == 0)
        error("vsv_make_server failure\n");
    if (vsv_up_connections(30))
        error("vsv_up_connections failure\n");
}

```

```

// wait for ceti to unlock me
semaphore_get(WAIT, semaphore_id, 1);
printf("rigel is done\n");
vsv_close_conn(connection);
}

```

16.4.2 Calling a C Function from a Vera Program Example

This example demonstrates how a Vera program running on host named "rigel" calls a function in a C program running on a host named "ceti".

Vera Program Running on "rigel"

```

#include <vera_defines.vrh>
program rigel
{
    integer connection, product;

    // make this program a client
    connection = vsv_make_client("ceti", 3333, 0);
    if (connection == 0)
        error("vsv_make_client failure\n");
    if (vsv_up_connections(30))
        error("vsv_up_connections failure\n");

    // call the remote multiply function on ceti
    if (vsv_call_func(connection, WAIT, "multiply", product, 8, 9))
        error("vsv_call_func failure\n");
    printf("product = %d\n", product);
    vsv_close_conn(connection);
}

```

C Program Running on "ceti"

```

#include <vsv.h>

static int done = 0;
static void multiply(vsvArgT *returnVal, int numberArgs, vsvArgT *args)
{
    /* set the return value */
    returnVal->data.integerVal =
        args[0].data.integerVal * args[1].data.integerVal;
    done = 1;
}

```

```

main()
{
    vsvArgT returnVal, args[2];
    void *connection;

    /* make this program a server */
    connection = vsv_MakeServer(3333,0);
    if(connection == NULL)
        perror(vsv_GetErrorMesg());

    /* define the argument types accepted by the multiply function */
    returnVal.vtype = VSV_VAR_ARG;
    returnVal.dtype = VSV_INTEGER_ARG;
    args[0].vtype = VSV_NONVAR_ARG;
    args[0].dtype = VSV_INTEGER_ARG;
    args[1].vtype = VSV_NONVAR_ARG;
    args[1].dtype = VSV_INTEGER_ARG;
    if(vsv_RegisterFunc(connection, "multiply", &returnVal, 2, args, multiply))
        perror(vsv_GetErrorMesg());
    if(vsv_UpConnections(30))
        perror(vsv_GetErrorMesg());

    /* wait for multiply to be called once */
    while(!done)
    {
        if(vsv_AnswerCalls(connection, VSV_DO_PENDING))
            perror(vsv_GetErrorMesg());
        sleep(1);
    }
    (void)vsv_CloseConn(connection);
    exit(0);
}

```

16.5 Troubleshooting

If the `vsv_up_connections()` system function is failing, use the `VERBOSE` option with `vsv_make_server()` and make sure that zombie UNIX/NT processes are not still running from previous simulations. You may also want to increase the timeout parameter passed to `vsv_up_connections()`.

If `vsv_up_connections()` works, but your network is slow and your remote calls are failing with errors such as "Connection timeout, no response," try setting these environment variables:

```

setenv VSV_READ_TIMEOUT 600
setenv VSV_BLOCKED_TIMEOUT 120

```

16.6 Backward Compatibility

For backward compatibility, these system function names from previous versions of Vera are still in use with their corresponding current names:

Table 16-1 Previous and Current VSV System Functions

Previous	Current
call_func	vsv_call_func
call_task	vsv_call_task
close_conn	vsv_close_conn
get_conn_err	vsv_get_conn_err
make_client	vsv_make_client
make_server	vsv_make_server
up_connections	vsv_up_connections

17. Testbench Setup and Usage Notes

This chapter discusses general setup and usage notes for creating and running simulations with Vera testbenches. It includes these sections:

- Creating Testbenches for Verilog Designs
- Creating Testbenches for VHDL Designs
- VHDL Testbench Usage Notes

17.1 Creating Testbenches for Verilog Designs

This section discusses the creation of Vera testbenches using the Vera template generator. It also details how to connect Vera testbenches to Verilog designs through the test-top file and through direct connections. Finally, it details how to set up multiple clocking domains.

17.1.1 Vera Template Generator

Vera's template generator creates a simulation framework and a Vera program template from an HDL module definition. The syntax to invoke the template generator is:

```
vera -tem template_options HDL_filename
```

template_options - The *template_options* are listed in Table 17-1.

Table 17-1 Template Options

Option	Definition
-c	Specifies clock signal
-d	Specifies dump file
-nav	Uses Magellan for simulation control
-t	Specifies Verilog module name

HDL_filename - The *HDL_filename* is the HDL file from which the template is generated.

-c

The **-c** option specifies the clock signal on the HDL module. The syntax is:

```
-c clock_signal
```

clock_signal - The *clock_signal* is the signal you want to connect SystemClock to.

When the `-c` option is invoked, the defined clock signal is connected to `SystemClock`, and it is specified as `CLOCK` in the interface specification. If this option is omitted, `SystemClock` is used as the sampling clock.

-d

The `-d` option specifies the dump file name. The syntax is:

```
-d filename
```

filename - The *filename* is the name of the dump file.

-nav

The `-nav` option invokes Magellan for simulation control, source level debugging, and waveform display. The syntax is:

```
-nav
```

To use Magellan or any other graphic package, you need the corresponding PLI tasks linked to your HDL executable.

-t

The `-t` option specifies the top HDL module name. The syntax is:

```
-t filename
```

filename - The *filename* is the name of the top HDL module from which the template is being generated.

The `-t` option is mandatory. It is used to generate three output files: *filename.test_top.v*, *filename.if.vrh*, and *filename.vr.tmp*.

This is an example module from a Verilog model file *sample.v*:

```
module myfsm(isig, osig, others, Clk);
  input isig;
  output osig;
  inout [3:0] others;
  input Clk;
  ...
endmodule
```

Given this Verilog model and module, this command line is invoked to generate a Vera template:

```
vera -tem -t myfsm -c Clk -nav -d myfsm.dump sample.v
```

When the command line is invoked, this test top file *myfsm.test_top.v* is generated, which contains the instances of `vera_shell` and the specified module with all signal connections:


```

module myfsm_test_top;
    parameter simulation_cycle = 100;
    reg SystemClock;
    wire isig;
    wire osig;
    wire [3:0] others;
    wire Clk;
    assign Clk = SystemClock;

    vera_shell vshell(
        .SystemClock(SystemClock),
        .myfsm_isig(isig),
        .myfsm_osig(osig),
        .myfsm_others(others),
        .myfsm_Clk(Clk)
    );

    myfsm dut(
        .isig(isig),
        .osig(osig),
        .others(others),
        .Clk(Clk)
    );

    initial begin
        SystemClock = 0;
        forever begin
            #(simulation_cycle/2)
            SystemClock = ~SystemClock;
        end
    end

    initial begin
        $dumpfile("myfsm.dump");
        $dumpvars(0,myfsm_test_top);
    end

endmodule

```

Invoking the command line also generates the interface file *myfsm.if.vrh*:

```

interface myfsm {
    output isig OUTPUT_EDGE OUTPUT_SKEW;
    input osig INPUT_EDGE;
    inout [3:0]others INPUT_EDGE OUTPUT_EDGE OUTPUT_SKEW;
    input Clk CLOCK;
}

```

Finally, the command line generates the Vera program template file *myfsm.vr.tmp*:

```
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>

program myfsm_test
    // declaring interface, vca signal, and classes..

#include "myfsm.if.vrh"
    // define vca_defn, class, instance here if necessary

{ // Start of myfsm_test
    ...
} // end of program myfsm_test
```

17.1.2 Connecting Vera Testbenches to Verilog Designs

There are three ways to connect Vera testbenches to Verilog designs: through a test-top file, by driving Verilog submodules through direct signal connections, and through the multiple module support discussed in Chapter 12.

17.1.2.1 Connecting Testbenches Through the Test-top File

The test-top file method of connecting testbenches to the DUT connects the Vera testbench to the Verilog DUT through a test-top. The DUT and the Vera shell file (.vshell) rest on the test-top, and all signal connections are made through that test-top. The test-top instantiates the DUT and the Vera shell module. Vera drives the Verilog DUT through the shell module. Finally, the clock generator is on the test-top. Figure 17-1 shows the schematic of this configuration.

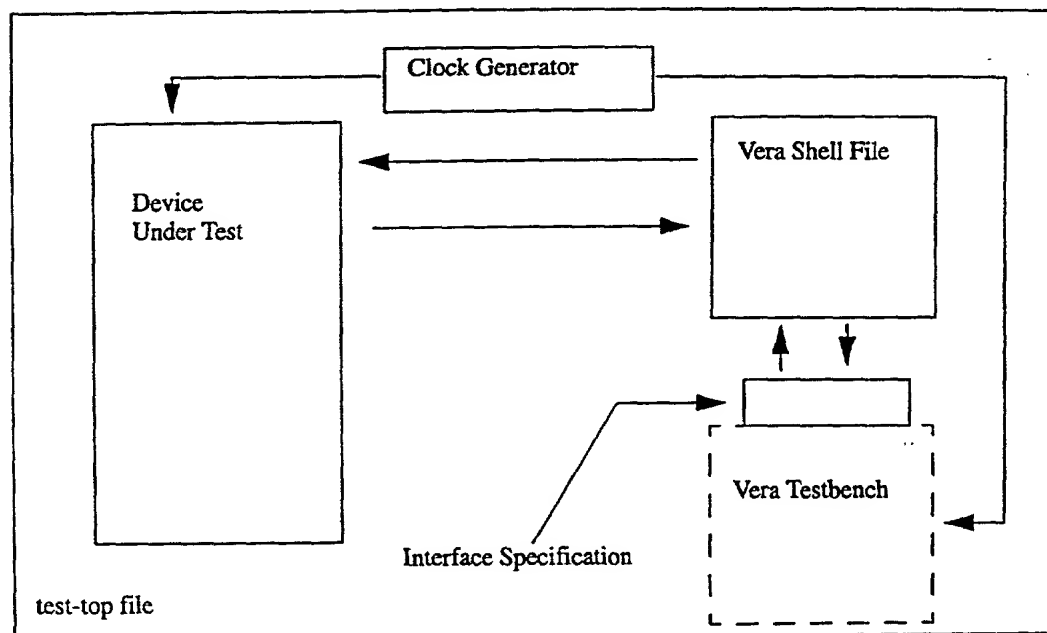


Figure 17-1 Test-top method of testbench connection

Given this setup, the steps to connect a Vera testbench to a Verilog arbiter (*rrarb.v*) are:

1. Generate the template files from the Verilog source code using:

```
vera -tem -t rrarb -c clk rrarb.v
```

This generates the interface file and the test-top file. The interface file specifies the signal connections between the DUT and the testbench. If you use the same signal names in the Vera testbench as the names in the DUT, you do not need to modify the interface file.

The clock generator is created in the test-top file.

2. Compile the Vera testbench source code using:

```
- vera -cmp rrarb.vr
```

This generates the compiled Vera object files (.vro) and the Vera shell file (.vshell).

3. Run the simulation using:

```
vera -run verilog rrarb.test_top.v rrarb.v rrarb.vshell \
+vera_prog=rrarb.vro
```

Using this configuration, these steps connect the testbench and DUT without any further work.

The files for this arbiter are:

- *rrarb.v* (the Verilog design)
- *rrarb.vr* (the Vera testbench)

The DUT design generates these files:

- *rrarb.if.vrh* (the interface file)
- *rrarb.test_top.v* (the test-top file)
- *rrarb.vshell* (the Vera shell file)
- *rrarb.vro* (Vera object file)

rrarb.v

```
module rrarb (request, grant, reset, clk);
    input [1:0] request;
    output [1:0] grant;
    input reset;
    input clk;
    wire winner;
    reg last_winner;
    reg [1:0] grant;
    wire [1:0] next_grant;

    assign next_grant[0]
        = ~reset & ( request[0] &
            (~request[1] | last_winner) );

    assign next_grant[1]
        = ~reset & ( request[1] &
            (~request[0] | ~last_winner) );

    assign winner
        = ~reset & ~next_grant[0] &
            ( last_winner | next_grant[1] );

    always @(posedge clk) begin
        last_winner = winner ;
        grant = next_grant ;
    end
endmodule
```

rrarb.vr

```
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>
#include "rrarb.if.vrh"

program rrarb_test
{
    integer i;

    trace( ON, PROGRAM, 0 );
// reset
    rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b0;
    rrarb.reset = 1;
    @1 rrarb.reset = 0;
// simple requesting
    @1 rrarb.request[0] = 1'b1;
    @1 rrarb.grant == 2'b01;
    @1 rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b1;
    @1 rrarb.grant == 2'b10;
    rrarb.request[1] = 1'b0;
// round-robin test
    @1 rrarb.request[0] = 1'b1;
    rrarb.request[1] = 1'b1;
    @1 rrarb.grant == 2'b01; // should get grant 0, since last was 1
    @1 rrarb.grant == 2'b10; // keep asserting both port, should get grant 1
    rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b0;
} // end of program rrarb_test
```

17.1.2.2 Directly Connecting Testbenches to the DUT

Direct connections to the DUT eliminate the need for a test-top file. Instead, the testbench is directly connected to the DUT through the HDL node method of signal declarations. Using this configuration, you must explicitly define how the testbench signals and DUT signals are connected. It is important to note that under this configuration, the clock must be defined in the DUT and connected to the testbench through a signal declaration. Figure 17-2 shows the schematic for this configuration.

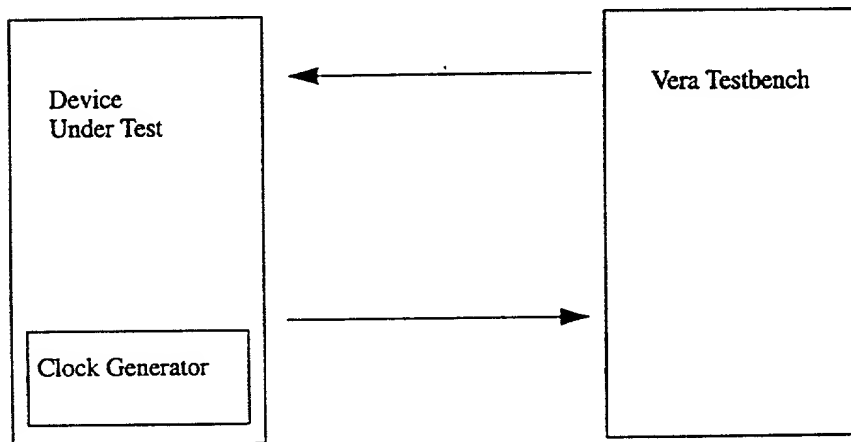


Figure 17-2 Direct connection between testbench and DUT

Given this setup, the steps to connect a Vera testbench to a Verilog arbiter (*rrarb.v*) are:

1. Compile the Vera testbench source code, including the interface specification, using:

```
vera -cmp rrarb.vr
```

The interface specification is generally included in the *rrarb.if.vrh* file and included in the *rrarb.vr* file. This generates the compiled Vera object files (.vro) and the Vera shell file (.vshell).

2. Run the simulation using:

```
vera -run verilog rrarb.v rrarb.vshell +vera_prog=rrarb.vro
```

Using this configuration, the clock must be defined in the DUT and connected to the testbench through the interface specification. Further, each signal connection must be done using the hierarchal pathnames in the HDL node method.

The files for this arbiter are:

- *rrarb.v* (the Verilog design)
- *rrarb.vr* (the Vera testbench)
- *rrarb.if.vrh* (the interface file)

The DUT design generates these files:

- *rrarb.vshell* (the Vera shell file)
- *rrarb.vro* (Vera object file)

rrarb.v

```

module rrarb;
    parameter simulation_cycle = 100 ;
    wire [1:0] request ;
    wire reset ;
    reg clk ;
    wire winner ;
    reg last_winner ;
    reg [1:0] grant ;
    wire [1:0] next_grant ;

    initial
        begin
            clk = 0 ;
            forever begin
                #(simulation_cycle/2)
                clk = ~clk ;
            end
        end

    assign next_grant[0]
        = ~reset & ( request[0] & (~request[1] | last_winner) ) ;
    assign next_grant[1]
        = ~reset & ( request[1] & (~request[0] | ~last_winner) ) ;
    assign winner
        = ~reset & ~next_grant[0] & ( last_winner | next_grant[1] ) ;
    always @(posedge clk) begin
        last_winner = winner ;
        grant = next_grant ;
    end
endmodule

```

rrarb.if.vrh

```

#ifndef INC_RRARB_IF_VRH
#define INC_RRARB_IF_VRH

interface rrarb
{
    output[1:0] request OUTPUT_EDGE OUTPUT_SKEW verilog_node
        "rrarb.request";
    input [1:0] grant INPUT_EDGE verilog_node "rrarb.grant";
    output reset OUTPUT_EDGE OUTPUT_SKEW verilog_node "rrarb.reset";
    input clk CLOCK verilog_node "rrarb.clk";
}
verilog_node CLOCK "rrarb.clk";
#endif

```

rrarb.vr

```

#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>
#include "rrarb.if.vrh"

program rrarb_test
{
    integer i;

    trace( ON, PROGRAM, 0 );

    // reset
    rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b0;
    rrarb.reset = 1;
    @1 rrarb.reset = 0;

    // simple requesting
    @1 rrarb.request[0] = 1'b1;
    @1 rrarb.grant == 2'b01;
    @1 rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b1;
    @1 rrarb.grant == 2'b10;
    rrarb.request[1] = 1'b0;

    // round-robin test
    @1 rrarb.request[0] = 1'b1;
    rrarb.request[1] = 1'b1;
    @1 rrarb.grant == 2'b01; // should get grant 0, since last was 1
    @1 rrarb.grant == 2'b10; // keep asserting both port, should get grant 1

```



```

rrarb.request[0] = 1'b0;
rrarb.request[1] = 1'b0;
} // end of program rrarb_test

```

17.1.3 Multiple Clocking Domains

There are two ways to run a simulation with multiple clock domains: create multiple clock generators in the DUT source files, or create multiple clock generators in the test-top file.

If you want to use multiple clocking domains with the test_top generators, you should add these lines of code after the SystemClock declaration in the test-top file:

```

initial begin
    clockname = 0;
    forever begin
        # period/2
        clockname = ~clockname
    end
end

```

clockname - The *clockname* is the name of the clock generator.

period - The *period* is the period of a single clock cycle, measured in nanoseconds.

17.2 Creating Testbenches for VHDL Designs

To integrate Vera testbenches with VHDL designs, you must develop the simulation framework. This section includes discussions on each of the files required to integrate Vera testbenches with VHDL designs and how they are made, instructions on how to connect Vera testbenches to VHDL designs, and instructions on how to generate multiple clocking domains.

17.2.1 Files for Test-top Connections

VHDL's version of the test-top connection requires three key files: *design.top*, *design_top.vhd*, *design_inf.vhd* and *vera.ini*. Each of these files is discussed here.

17.2.1.1 *design.top*

The *design.top* file is the test top file used to connect the VHDL design to the testbench. To generate the top file, use the -vhd1 or -vhd1 -top compile options:

```
vera -cmp -vhd1 -top filename
```

filename - The *filename* is the name of the Vera testbench source file. This file contains the interface specification for the testbench.

Using the `-top` switch with the `-vhdl` switch generates the top file. When the file is generated, you must rename it to *filename.top*.

After generating the top file, you must modify it:

1. Instantiate the DUT. You must define the DUT component.
2. Create a DUT instance with a valid port map.
3. Define the signals referenced by the port map.
4. If the DUT has a system clock, connect the DUT's system clock to the `SystemClock` signal. If the DUT does not have a system clock, modify the generated system clock's half period as necessary for your simulation.

Warning – Do NOT modify the `vmc_mti` entity or architecture.

The top file is a source file that is modified only if the Vera interface changes (signals are added, deleted, or renamed) or if the DUT interface changes. If neither changes, the top file does not need to be modified or regenerated.

17.2.1.2 *design_top.vhd* and *design_intf.vhd*

The *design_top.vhd* file is generated using the `-vhdl` switch alone:

```
vera -cmp -vhdl filename
```

When the `-vhdl` switch is used alone, the *filename_intf.vhd* and *filename_top.vhd* files are generated. These files should not be renamed or modified. If you need to modify the *design_top.vhd* file, edit the *design.top* file. These files must be compiled with the VHDL source files. The `-vhdl` switch checks for the existence of a *filename.top* file. If the file does not exist, a warning is issued and the *filename_top.vhd* file is not generated.

VHDL tasks to be called from Vera should be encapsulated and included in the *filename_top.vhd* file.

Vera's global variables are copied into the *filename_top.vhd* file so that they can be viewed and debugged. They are prepended by `var_`. Only Vera variables of type integer, bit, and bit[] are copied.

Inside the *filename_top.vhd* file is a reference to the Vera shared library *vera_mti.dll*. For VSIM to find this file, you must set `LD_LIBRARY_PATH` to `$VERA_HOME/lib`.

17.2.1.3 *vera.ini*

The initialization file *vera.ini* is required for the VSIM command. Vera first checks the current working directory and then the `$VERA_HOME` directory. This file must be present for Vera to run.

The *vera.ini* file contains lines of this syntax:

name_token value_token

name_token - The valid *name_tokens* are:

vera_load
vera_mload
vera_mload_define
vera_stop_on_error
vera_continue_on_error
vera_exit_on_error
vera_finish_on_end
vera_debug_on_error
vera_text_debug
vera_semaphore_size
vera_region_size
vera_mailbox_size
vera_random_seed
vera_rand48_seed
map

value_token - The *value_token* varies depending on the *name_token*. The valid tokens are listed in the subsequent definitions of the *name_tokens*.

vera_load

The *vera_load* token specifies an HDL object (.vro) file to be used with the simulation. The syntax is:

vera_load filename

filename - The *filename* is the .vro object file to be used. Multiple files can be specified by separating them with commas.

vera_mload

The *vera_mload* token specifies a .vrl file that lists multiple object files to be used with the simulation. The syntax is:

vera_mload filename

filename - The *filename* is the .vrl file that lists the object files to be used. Multiple files can be specified by separating them with commas.

vera_mload_define

The *vera_mload_define* specifies a text macro. The syntax is

vera_mload_define macro_name=value

macro_name - The *macro_name* is the string of text to be substituted for.

value - The text macro is optionally set to *value*.

Text macros defined at compile time are particularly useful when using the conditional compilation directives.

vera_stop_on_error

The **vera_stop_on_error** token causes the simulation to stop immediately when a simulation error is encountered. The syntax is:

```
vera_stop_on_error switch
```

switch - The *switch* is used to turn the mode on. A value of 1, "true," "on," or "yes" turns the mode on.

The default setting is to execute the remaining code within the present simulation time.

vera_continue_on_error

The **vera_continue_on_error** causes the simulation to continue after a verification error is encountered. The syntax is:

```
vera_continue_on_error switch
```

switch - The *switch* is used to turn the mode on. A value of 1, "true," "on," or "yes" turns the mode on.

This token causes the simulation to continue when verification errors (such as failed expects) occur. This token overrides the **vera_exit_on_error** token.

vera_exit_on_error

The **vera_exit_on_error** token causes the simulation to be terminated when a simulation error is encountered. The syntax is:

```
vera_exit_on_error switch
```

switch - The *switch* is used to turn the mode on. A value of 1, "true," "on," or "yes" turns the mode on.

The default setting is to stop the simulation and return to the HDL shell.

vera_finish_on_end

The **vera_finish_on_end** token specifies that the simulation is finished when a simulation run is completed. The syntax is:

```
vera_finish_on_end switch
```

switch - The *switch* is used to turn the mode on. A value of 1, "true," "on," or "yes" turns the mode on.

The default setting is to stop the simulation and return to the HDL shell.

vera_debug_on_error

The **vera_debug_on_error** token invokes a debugger when a verification error (such as a failed expect) is encountered. The syntax is:

```
vera_debug_on_error switch
```

switch - The *switch* is used to turn the mode on. A value of 1, "true," "on," or "yes" turns the mode on.

The **vera_debug_on_error** token determines the debugger used when a verification error is encountered. If the HDL simulator has control when the error occurs, the simulator's debugger is invoked. If Vera has control when the error occurs, Vera's debugger is invoked.

If the **vera_debug_on_error** token is not used, the behavior depends on the particular HDL simulator.

vera_text_debug

The **vera_text_debug** token launches the text debugger when an error occurs. The syntax is:

```
vera_text_debug switch
```

switch - The *switch* is used to turn the mode on. A value of 1, "true," "on," or "yes" turns the mode on.

vera_semaphore_size

The **vera_semaphore_size** token specifies the maximum number of semaphores within the simulation. The syntax is:

```
vera_semaphore_size size
```

size - The *size* specifies the maximum number of semaphores within the simulation.

The default limit is 2048.

vera_region_size

The **vera_region_size** token specifies the maximum number of regions within the simulation. The syntax is:

```
vera_region_size size
```

size - The *size* specifies the maximum number of regions within the simulation.

The default limit is 256.

vera_mailbox_size

The **vera_mailbox_size** token specifies the maximum number of mailboxes within the simulation. The syntax is:

```
vera_mailboxes_size size
```

size - The *size* specifies the maximum number of mailboxes within the simulation.

The default limit is 256.

vera_random_seed

The **vera_random_seed** token sets the seed for calls to the **random()** system function. The syntax is:

```
vera_random_seed seed
```

seed - The *seed* can be any valid expression evaluating to a number.

vera_rand48_seed

The **vera_random_seed** token sets the seed for calls to the **rand48()** system function. The syntax is:

```
vera_rand48_seed seed
```

seed - The *seed* can be any valid expression evaluating to a number.

map

The **map** token maps Vera enumerated types to VHDL enumerated types. The syntax is:

```
map vera_enum_name num vhd1_enum1 vhd1_enum2 ... vhd1_enumN
```

vera_enum_name - The *vera_enum_name* is the Vera enumerated type category.

num - The *num* specifies how many elements there are in the category.

vhd1_enumN - The *vhd1_enumN* is the VHDL enumerated type to which the Vera type is mapped.

When the **map** token is executed, each Vera enumerated type (up the specified number) is sequentially mapped to the specified VHDL name.

For example, in Vera, an enumerated type is defined as:

```
enum vera_color = VERA_RED, VERA_YELLOW, VERA_GREEN, VERA_BLUE;
```

In VHDL, an enumerated type is defined as:

```
type Vhd1Color is (VHDL_RED, VHDL_YELLOW, VHDL_GREEN, VHDL_BLUE);
```

To equate the two, use the **map** command:

```
map vera_color 4 VHDL_RED VHDL_YELLOW VHDL_GREEN VHDL_BLUE
```

17.2.1.4 vera

The **vera** command is also supported by VSIM:

```
vera token_name token_value
```

These token and value combinations are allowed:

```
debug
load filename
mload filename
```

debug

The **debug** token launches the Vera debugger when invoked. The syntax is:

```
vera debug switch
```

switch - The *switch* is used to turn the mode on. A value of 1, "true," "on," or "yes" turns the mode on.

load

The **load** token specifies an HDL object (.vro) file to be used with the simulation. The syntax is:

```
vera load filename
```

filename - The *filename* is the .vro object file to be used. Multiple files can be specified by separating them with commas.

mload

The **mload** token specifies a .vrl file that lists multiple object files to be used with the simulation. The syntax is:

```
vera mload filename
```

filename - The *filename* is the .vrl file that lists the object files to be used. Multiple files can be specified by separating them with commas.

17.2.1.5 VSIM Commands in vera.ini

Vera supports the VSIM Restart command. However, the Save and Restore commands are not supported.

17.2.2 Connecting Vera Testbenches to VHDL Designs

You can connect Vera testbenches to VHDL designs using two methods: connecting the testbench through a test-top configuration, or directly connecting the testbench to nodes within the design.

17.2.2.1 Connecting Testbenches Through the Test-top

The test-top method of connecting a Vera testbench to a VHDL design connects the testbench through the test-top file (*design_top.vhd*). The test-top file is generated from the *design.top* file. The DUT is connected directly to the Vera testbench through the Vera interface and through the VHDL interface. Figure 17-3 shows the schematic for this configuration.

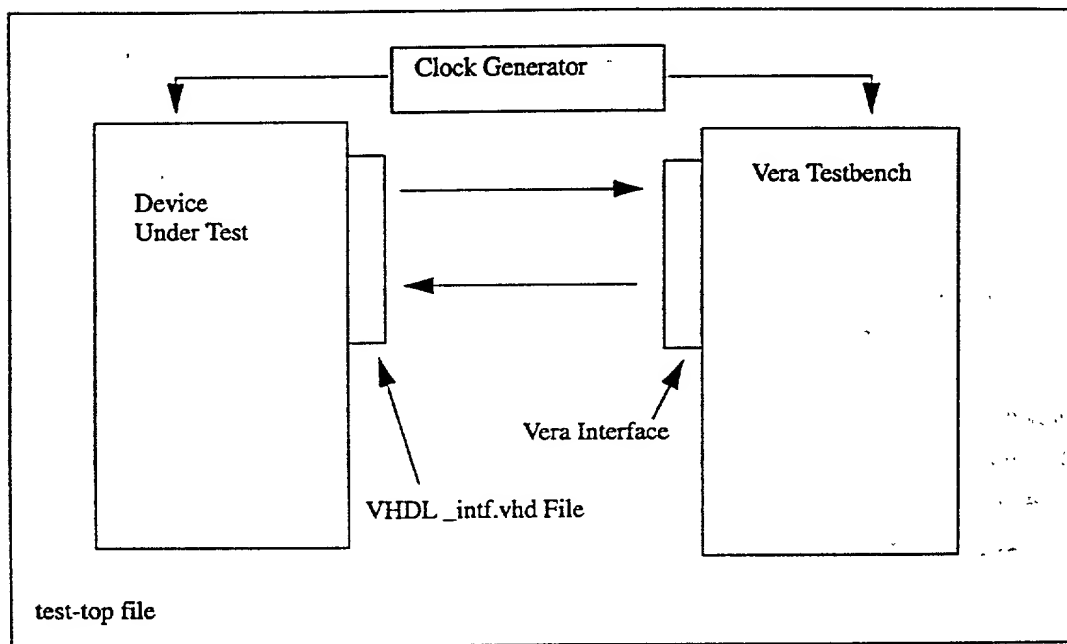


Figure 17-3 Test-top method of testbench connection

Under this configuration, the clock generator is in the test-top (*design_top.vhd*) file.

Given this setup, the steps to connect a Vera testbench to a VHDL arbiter (*rrarb.vhd*) are:

1. Generate the *rrarb.top* file from the Vera source code using:

```
vera -cmp -vhdl -top rrarb.vr
```

This generates the *rrarb.top* file.

2. Edit the *rrarb.top* file to include the definition and instantiation of the DUT.
3. Generate the VHDL interface file (*rrarb_intf.vhd*), test-top file (*rrarb_top.vhd*), and compiled Vera object file (*rrarb.vro*) using:

```
vera -cmp -vhdl rrarb.vr
```


This generates the interface file, test-top file, and compiled Vera object file from the Vera source code and the previously generated *rrarb.top* file. These files should not be edited. If you edit the *rrarb_top.vhd* file and recompile the Vera source code with the *-vhd* switch, you lose the edits you made.

4. Run the simulation with the VHDL interface file (*rrarb_intf.vhd*), the test-top file (*rrarb_top.vhd*), the Vera object file (*rrarb.vro*) and the VHDL DUT (*rrarb.vhd*).

The files from this arbiter are:

- *rrarb.vhd* (the VHDL design)
- *rrarb.if.vr* (the Vera interface file)
- *rrarb.vr* (the Vera testbench)

When compiled, these files are generated:

- *rrarb_intf.vhd* (the VHDL interface file)
- *rrarb_top.vhd* (the test-top file)
- *rrarb.vro* (Vera object file)

rrarb.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity rrarb is
    port (
        request : in  std_logic_vector(1 downto 0);
        grant   : out std_logic_vector(1 downto 0);
        reset   : in  std_logic;
        clk     : in  std_logic
    );
end entity rrarb;

architecture rrarb_arch of rrarb is
    signal winner : std_logic := '0';
    signal last_winner : std_logic := '0';
    signal next_grant : std_logic_vector(1 downto 0) := "00";
begin
    next_grant(0) <= (not reset) and ( request(0) and
        ((not request(1)) or last_winner) ) ;
    next_grant(1) <= (not reset) and ( request(1) and
        ((not request(0)) or (not last_winner) ) ) ;
    winner <= (not reset) and (not next_grant(0)) and
        (last_winner or next_grant(1));
    process (clk) is
    begin
```

```

        if (clk'event and clk = '1') then
            last_winner <= winner;
            grant <= next_grant;
        end if;
    end process;
end rrarb_arch;

```

rrarb.if.vrh

```

#ifndef INC_RRARB_IF_VRH
#define INC_RRARB_IF_VRH

interface rrarb
{
    output [1:0] request OUTPUT_EDGE OUTPUT_SKEW;
    input [1:0] grant INPUT_EDGE;
    output reset OUTPUT_EDGE OUTPUT_SKEW;
    input clk CLOCK;
} // end of interface rrarb
#endif

```

rrarb.vr

```

#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>
#include "rrarb.if.vrh"

program rrarb_test
{
    integer i;

    trace( ON, PROGRAM, 0 );
// reset
    rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b0;
    rrarb.reset = 1;
    @1 rrarb.reset = 0;
// simple requesting
    @1 rrarb.request[0] = 1'b1;
    @1 rrarb.grant == 2'b01;
    @1 rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b1;
    @1 rrarb.grant == 2'b10;
    rrarb.request[1] = 1'b0;
// round-robin test

```

```

@1 rrarb.request[0] = 1'b1;
rrarb.request[1] = 1'b1;
@1 rrarb.grant == 2'b01; // should get grant 0, since last was 1
@1 rrarb.grant == 2'b10; // keep asserting both port, should get grant 1
rrarb.request[0] = 1'b0;
rrarb.request[1] = 1'b0;
} // end of program rrarb_test

```

rrarb.top (after being modified)

```

library IEEE;
use IEEE.std_logic_1164.all;

entity rrarb_bench is
end;

architecture rrarb_bench_arch of rrarb_bench is
    #include rrarb.inc_vhd SIGNALS
    component rrarb
        port (
            request : in std_logic_vector(1 downto 0);
            grant : out std_logic_vector(1 downto 0);
            reset : in std_logic;
            clk : in std_logic
        );
    end component;
    component vmc_mti
    end component;
    begin
        system_clock <= not system_clock after 50 ns;
        rrarb_clk <= system_clock;
        rrarb_inst: component rrarb
            port map (
                request => rrarb_request,
                grant => rrarb_grant,
                reset => rrarb_reset,
                clk => rrarb_clk
            );
        vmc_mti_inst: vmc_mti;
        #include rrarb.inc_vhd TASKS
    end;

```

17.2.3 Directly Connecting Testbenches to the DUT

For VHDL, the model for directly connecting testbenches to the DUT is identical to using the test-top method, except for how the signals are connected. To directly connect the testbench to the DUT, you must use the HDL node method when specifying the interface signals.

The same arbiter from the previous section is included here with the connection made through direct nodes. Note the *rrarb.if.vr* file in particular.

rrarb.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity rrarb is
end entity rrarb;

architecture rrarb_arch of rrarb is
    signal winner : std_logic := '0';
    signal last_winner : std_logic := '0';
    signal reset : std_logic := '0';
    signal clk : std_logic := '0';
    signal next_grant : std_logic_vector(1 downto 0) := "00";
    signal grant : std_logic_vector(1 downto 0) := "00";
    signal request : std_logic_vector(1 downto 0) := "00";
begin
    clk <= not clk after 50 ns;
    next_grant(0) <= (not reset) and ( request(0) and
        ((not request(1)) or last_winner) ) ;
    next_grant(1) <= (not reset) and ( request(1) and
        ((not request(0)) or (not last_winner) ) ) ;
    winner <= (not reset) and (not next_grant(0)) and
        (last_winner or next_grant(1));
    process (clk) is
    begin
        if (clk'event and clk = '1') then
            last_winner <= winner;
            grant <= next_grant;
        end if
    end process;
end rrarb_arch;
```

rrarb.if.vrh

```

#ifndef INC_RRARB_IF_VRH
#define INC_RRARB_IF_VRH

interface rrarb
{
    output [1:0] request OUTPUT_EDGE OUTPUT_SKEW vhd1_node
        "/rrarb_bench/rrarb_inst/request";
    input [1:0] grant INPUT_EDGE vhd1_node
        "/rrarb_bench/rrarb_inst/grant";
    output reset OUTPUT_EDGE OUTPUT_SKEW vhd1_node
        "/rrarb_bench/rrarb_inst/reset";
    input clk CLOCK vhd1_node "/rrarb_bench/rrarb_inst/clk";
} // end of interface rrarb
vhd1_node CLOCK "/rrarb_bench/rrarb_inst/clk";
#endif

```

rrarb.vr

```

#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
#include <vera_defines.vrh>
#include "rrarb.if.vrh"

program rrarb_test
{
    integer i;

    trace( ON, PROGRAM, 0 );
    // reset
    rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b0;
    rrarb.reset = 1;
    @1 rrarb.reset = 0;
    // simple requesting
    @1 rrarb.request[0] = 1'b1;
    @1 rrarb.grant == 2'b01;
    @1 rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b1;
    @1 rrarb.grant == 2'b10;
    rrarb.request[1] = 1'b0;
    // round-robin test
    @1 rrarb.request[0] = 1'b1;
    rrarb.request[1] = 1'b1;
    @1 rrarb.grant == 2'b01; // should get grant 0, since last was 1

```

```

    @1 rrarb.grant == 2'b10; //keep asserting both port, should get grant 1
    rrarb.request[0] = 1'b0;
    rrarb.request[1] = 1'b0;
} // end of program rrarb_test

```

rrarb.top (after being modified)

```

library IEEE;
use IEEE.std_logic_1164.all;

entity rrarb_bench is
end;

architecture rrarb_bench_arch of rrarb_bench is
    #include rrarb.inc_vhd SIGNALS
    component rrarb
    end component;
    component vmc_mti
    end component;
    begin
        system_clock <= not system_clock after 50 ns;
        rrarb_clk <= system_clock;
        rrarb_inst: rrarb;
        vmc_mti_inst: vmc_mti;
        #include rrarb.inc_vhd TASKS
    end;

```

17.2.4 Multiple Clocking Domains

To run simulations with multiple clocking domains, you must:

1. Define an interface for each clocking domain you want to have in the interface file.
2. Modify the *filename_top.vhd* file to instantiate the clock:

```
-- signal signal_name : std_logic := '0';
```

3. Modify the *filename_top.vhd* file to connect the clock and set the half period:

```
clock_name <= not clock_name after period ns;
```

Note – You should have a clock for each interface.

17.3 VHDL Testbench Usage Notes

This section includes several usage notes on using VHDL testbenches.

17.3.1 Valid VHDL Signal Types

Vera supports the assertion and querying of only these VHDL signal types:

- `integer`
- `bit`
- `bit_vector`
- `std_logic`
- `std_logic_vector`
- `std_ulogic`
- `std_ulogic_vector`
- `boolean`
- `boolean array`
- `enumerated types`

17.3.2 VHDL Plus Arguments

In the *vera.ini* file, you can specify plus arguments. The syntax is:

`argument value`

argument - The *argument* is the name of the plus argument being defined.

value - The *value* is the value of the plus argument. There must be a space between the argument and the value.

For example, the VHDL equivalent to the Verilog plus argument `+vera_my_arg=100` is:

```
vera_my_arg= 100
```

Note the space between the argument name and the argument value.

17.3.3 Features Not Yet Supported for VHDL

These features are not yet supported for VHDL testbenches:

- Multiple module support
- Vera-CORE
- Calling Vera tasks from VHDL

Copyright © 2000-2001 Synopsys, Inc.

18. Compilation

The Vera compiler accepts a Vera source program and generates binary Vera objects. This chapter discusses the Vera compiler, preprocessor, and runtime control. It details Vera's use with different HDL simulators, with C functional simulators, and as a stand-alone cycle-based simulator. This chapter includes these sections:

- Compiler Overview
- Preprocessor
- General Options
- Compile Options
- Running Vera with HDLs
- Modular Compilation
- Dynamic Loading of Vera Object Modules
- Running Vera Stand-alone

18.1 Compiler Overview

The compiler options are listed with a brief description in Table 18-1.

Table 18-1 Vera Compiler Options and Definitions

Option	Definition
-help	Lists valid compiler options
-cmp	Compiles Vera source files
-alim	Sets maximum number of elements in arrays
-ansi	Invokes ansi preprocessor
-g	Invokes compiler using debug information
-h	Updates .vrh file
-hnu	Updates .vrh file only if it has changed
-i	Produces ASCII interface definition
-ip	Compiles an IP core
-max_error	Sets the maximum number of errors before compilation failure
-q	Compiles in quiet mode

Table 18-1 Vera Compiler Options and Definitions (Continued)

Option	Definition
-timescale	Sets timescale
-top	Creates VHDL top file
-vhdl	Processes VHDL top file
-D	Specifies text macro
-I	Includes directory in search path
-pp	Invokes preprocessor.
-proj	Creates Vera shell files from project file
-run	Runs simulation
-tem	Invokes Vera template generator
-c	Specifies clock signal
-d	Specifies dump file
-nav	Uses Magellan for simulation control
-t	Specifies Verilog module name
-v	Returns the compiler version number
-vcon	Generates a .vcon file

The options available for use with Vera-CS are listed with a brief description in Table 18-2.

Table 18-2 Vera-CS Options and Definitions

Option	Definition
-e	Exits simulation on error
-f	Includes .vrl file in the compile
-i	Specifies initialization file
-s	Pauses simulation on error
-t	Invokes text-based debugger

18.2 Preprocessor

Vera programs are processed by a custom preprocessor prior to being translated by the compiler. The preprocessor handles file inclusion, text macros, and conditional compilation.

18.2.1 File Inclusion

A Vera source file can include another Vera source file by reference using the include construct. The syntax is:

```
#include "path_to_file"
```

path_to_file - The *path_to_file* can be the absolute or relative pathname to the include file. If the file is in \$VERA_HOME/lib, the quotes can be replaced by angled brackets (<>), and only the file name is required.

The <vera_defines.vrh> file should always be included.

When the Vera compiler encounters an include statement, it replaces the include line with the entire file.

Note - You cannot have spaces before the # in include statements.

18.2.2 Text Macros

Vera also supports text macros. The syntax to define a text macro is:

```
#define macro_name macro_value;
```

macro_name - The *macro_name* is the name of the constant as it will be referenced throughout the Vera program.

macro_value - The *macro_value* is the value assigned to the *macro_name*. It must be a constant numeric or word substitution.

Note - You cannot have spaces before the # in text macros.

This is an example of text macros:

```
#define word_width 32
#define OUTPUT_EDGE PHOLD
#define OUTPUT_SKEW #1
#define INPUT_EDGE PSAMPLE
```

18.2.3 Conditional Compilation

The Vera preprocessor supports conditional compilation using #ifdef and #ifndef directives. The syntax is:

```
#ifdef macro_name
#ifdef macro_name
```

macro_name - The *macro_name* is the name of the text macro that determines the conditional.

Note - You cannot have spaces before the # in conditional directives.

The `ifdef` directive conditionally compiles the text after the directive if the specified text macro has been defined.

The `ifndef` directive conditionally compiles the text after the directive if the specified text macro has not been defined.

Each of these directives can be used with the `#else` directive. The syntax is:

```
#else
```

The `else` directive conditionally compiles the text after the directive if the previous `ifdef` or `ifndef` failed.

Conditional compilation should be terminated using the `#endif` directive. The syntax is:

```
#endif
```

This is an example of a conditional compilation directive block:

```
#ifdef MACRO1
    // This code is for MACRO1
    ...
#else
    // This code is not for MACRO1
    ...
#endif

#ifndef MACRO1
    // This code is not for BOO
    ...
#endif
```

18.3 General Options

When the Vera compiler is invoked, several options can be called. The syntax to invoke the compiler is:

```
vera general_options input_filename
```

general_options - The general options used with the Vera compiler and their definitions are listed in Table 18-3.

Table 18-3 General Options

Option	Definition
<code>-cmp</code>	Compiles Vera source code
<code>-I</code>	Includes directory in search path
<code>-pp</code>	Invokes preprocessor
<code>-proj</code>	Creates Vera shell files from project file

Table 18-3 General Options (Continued)

Option	Definition
-run	Runs simulation
-tem	Invokes Vera template generator
-v	Returns the compiler version number
-vcon	Generates a .vcon file

input_filename - The *input_filename* is the name of the Vera source file being processed.

-cmp

The **-cmp** option compiles Vera source code. It is discussed in detail in Section 18.4.

-I

The **-I** option adds a specified path to the search path for include files. The syntax is:

-I path_name

path_name - The *path_name* is the absolute or relative path to the directory you want to add to your search path.

The **-I** option adds the specified path to all include files. If this option is invoked, be sure to declare all include files with respect to the specified path.

-pp

The **-pp** option invokes the Vera preprocessor. The syntax is:

-pp

When the **-pp** option is used, the preprocessor is explicitly used on the input file. When using the **-cmp** option, the preprocessor is automatically invoked.

-proj

The **-proj** option is used when creating and using IP cores. The syntax is:

-proj

For more information on Vera-Core, see Chapter 13., "Vera-CORE."

-run

The **-run** option is used to run a Vera simulation with an HDL design. The syntax is:

-run

For more information on running simulations, see Section 18.5, "Running Vera with HDLs."

-tem

The **-tem** option invokes the Vera template generator. The syntax is:

-tem

The Vera template generator is discussed in more detail in Section 17.1.1, "Vera Template Generator."

-v

The **-v** option returns the version number of the Vera compiler. The syntax is:

-v

-vcon

The **-vcon** option generates a .vcon file for use with multiple module support. The syntax is:

-vcon

For more information on multiple module support, see Chapter 12., "Multiple Module Support."

18.4 Compile Options

The Vera compiler is primarily used to compile Vera source code using the **-cmp** option. The syntax is:

vera -cmp compiler_options input_filename output_filename

compiler_options - The compiler options and definitions are listed in Table 18-4.

Table 18-4 Vera Compiler Options

Option	Definition
-alim	Sets the maximum number of array elements
-ansi	Invokes the ansi preprocessor
-D	Specifies a macro name
-g	
-h	Updates the .vrh file
-hnu	Updates the .vrh file only if it has changed
-i	Produces an ASCII interface definition
-ip	Generates an IP core
-max_error	Sets the maximum number of errors before compilation failure

Table 18-4 Vera Compiler Options (Continued)

Option	Definition
-q	Compiles in quiet mode, without a header
-timescale	Specifies a timescale
-top	Generates a VHDL test_top file
-vhdl	Generates VHDL template files

input_filename - The *input_filename* is the name of the Vera source file being compiled.

output_filename - The *output_filename* is the prefix of the output file.

When the **-cmp** option is invoked, the compiler generates the object file *output_filename.vro* and the shell file *output_filename.vshell*. If the output file is not specified, the prefix of the input file is used. Compiler options are used on top of this basic function.

-alim

The **-alim** option explicitly sets the maximum number of elements allowed in arrays. The default limit is 1024. The syntax is:

-alim number

number - The maximum number of elements in an array is specified by *<number>*.

-ansi

The **-ansi** option preprocesses the Vera source code in ANSI mode. The syntax is:

-ansi

-D

The **-D** option specifies a text macro. The syntax is:

-D macro_name=value

macro_name - The *macro_name* is the string of text to be substituted for.

value - The text macro is optionally set to *value*.

Text macros defined at compile time are particularly useful when using the conditional compilation directives.

-g

The **-g** option generates debugging information. The syntax is:

-g

-h

The **-h** option updates the .vrh header file. The syntax is:

-h

The **-h** option automatically generates include files with global declarations from the Vera source code.

-hnu

The **-hnu** option updates the .vrh header file only if it has changed. The syntax is:

-hnu

The **-hnu** option functions like the **-h** option except that it only updates the .vrh file if there are changes to it.

-i

The **-i** option generates an ASCII file containing the interface definition. The syntax is:

-i

The **-i** option generates a .vri interface file from the Vera source code. This is interface specification produces the corresponding generated interface file:

```
input data_in[31:0] PSAMPLE 1
output ack[31:0] PHOLD 0
inout data_out[1:0] PSAMPLE 0 PHOLD 1

input if_0 data_in 32 PSAMPLE 1
output if_0 ack 32 PHOLD 0
inout if_0 data_out 1 PSAMPLE 0 PHOLD 1
```

-ip

The **-ip** generates Vera object files that can be used as IP cores. The syntax is:

-ip

When the **-ip** option is used, the Vera object files can be used without a runtime license. How IP cores are used and generated is discussed in detail in Chapter 13., "Vera-CORE."

-max_error

The **-max_error** option specifies how many errors are encountered before the compile fails. The syntax is:

-max_error number

number - The *number* specifies the maximum number of errors the compiler can encounter before the compilation fails.

-q

The **-q** option compiles the code in quiet mode. The syntax is:

```
-q
```

When the compiler is run in quiet mode, the header information is shortened and limited to only compile information.

-timescale

The **-timescale** option explicitly defines the simulation timescale inside the Vera shell file. The syntax is:

```
-timescale=scale
```

scale - The *scale* specifies the cycle length in nanoseconds.

The **-timescale** option generates this statement at the top of the Vera shell file:

```
`timescale scale
```

-top

The **-top** switch is used to generate a test_top file for VHDL designs. The syntax is:

```
-top
```

-vhdl

The **-vhdl** switch is used to generate template files for VHDL designs. The syntax is:

```
-vhdl
```

18.5 Running Vera with HDLs

Vera can be run with HDL designs using the **-run** option with plus arguments. The syntax is:

```
vera -run filename args
```

filename - The *filename* is the executable HDL file compiled with the Vera libraries.

args - The plus arguments and their definitions are listed in Table 18-5.

Table 18-5 Plus Arguments

Plus Argument	Definition
+vera_load	Specifies a single object file
+vera_mload	Specifies a file with a list of object files
+vera_stop_on_error	Execution stops on simulation error

Table 18-5 Plus Arguments (Continued)

Plus Argument	Definition
<code>+vera_continue_on_error</code>	Execution continues after verification error
<code>+vera_exit_on_error</code>	Exits simulator on error
<code>+vera_finish_on_end</code>	Exits simulator when simulation finishes
<code>+vera_debug_on_error</code>	Invokes Vera debugger on error
<code>+vera_text_debug</code>	Launches the Vera text debugger on error
<code>+vera_semaphore_size</code>	Specifies maximum number of semaphores
<code>+vera_region_size</code>	Specifies maximum number of regions
<code>+vera_mailbox_size</code>	Specifies maximum number of mailboxes

+vera_load

The `+vera_load` argument specifies an HDL object (.vro) file to be used with the simulation. The syntax is:

```
+vera_load=filename
```

filename - The *filename* is the .vro object file to be used. Multiple files can be specified by separating them with commas.

+vera_mload

The `+vera_mload` argument specifies a .vrl file that lists multiple object files to be used with the simulation. The syntax is:

```
+vera_mload=filename
```

filename - The *filename* is the .vrl file that lists the object files to be used. Multiple files can be specified by separating them with commas.

+vera_stop_on_error

The `+vera_stop_on_error` argument causes the simulation to stop immediately when a simulation error is encountered. The syntax is:

```
+vera_stop_on_error
```

The default setting is to execute the remaining code within the present simulation time.

+vera_continue_on_error

The **+vera_continue_on_error** causes the simulation to continue after a verification error is encountered. The syntax is:

```
+vera_continue_on_error
```

This argument causes the simulation to continue when verification errors (such as failed expects) occur. This argument overrides the **+vera_exit_on_error** argument.

+vera_exit_on_error

The **+vera_exit_on_error** argument causes the simulation to be terminated when a simulation error is encountered. The syntax is:

```
+vera_exit_on_error
```

The default setting is to stop the simulation and return to the HDL shell.

+vera_finish_on_end

The **+vera_finish_on_end** argument specifies that the simulation is finished when a simulation run is completed. The syntax is:

```
+vera_finish_on_end
```

The default setting is to stop the simulation and return to the HDL shell.

+vera_debug_on_error

The **+vera_debug_on_error** argument invokes a debugger when a verification error (such as a failed expect) is encountered. The syntax is:

```
+vera_debug_on_error
```

The **+vera_debug_on_error** argument determines the debugger used when a verification error is encountered. If the HDL simulator has control when the error occurs, the simulator's debugger is invoked. If Vera has control when the error occurs, Vera's debugger is invoked.

If the **+vera_debug_on_error** argument is not used, the behavior depends on the particular HDL simulator.

For example, with Verilog-XL, if the simulator has control, an interactive prompt is invoked. If Vera has control, the simulator invokes the interactive prompt when control passes back to the simulator.

With VCS, regardless of where control is, the interactive prompt is invoked.

+vera_text_debug

The **+vera_text_debug** argument is used to launch the Vera text-based debugger when an error is encountered. The syntax is:

```
+vera_debug_window
```

+vera_semaphore_size

The **+vera_semaphore_size** argument specifies the maximum number of semaphores within the simulation. The syntax is:

```
+vera_semaphore_size_size
```

size - The *size* specifies the maximum number of semaphores within the simulation.

The default limit is 2048.

+vera_region_size

The **+vera_region_size** argument specifies the maximum number of regions within the simulation. The syntax is:

```
+vera_region_size_size
```

size - The *size* specifies the maximum number of regions within the simulation.

The default limit is 256.

+vera_mailbox_size

The **+vera_mailbox_size** argument specifies the maximum number of mailboxes within the simulation. The syntax is:

```
+vera_mailboxes_size_size
```

size - The *size* specifies the maximum number of mailboxes within the simulation.

The default limit is 256.

18.6 Modular Compilation

Vera programs can be compiled modularly at the program, task, and function levels. This allows you to make changes to individual files and reload the changes. There is no need to recompile or reload the HDL design code when compiling modularly.

Modular compilation requires these conditions:

- Variables declared in the top block must be declared external if tasks and functions compiled separately reference them.
- Interface specifications in subroutine modules must match exactly those declared in the main program file.
- Naming conflicts with subroutines can be resolved using local subroutine names.

This is an example of two separate files linked at simulation time using modular compilation:

tst_chip.vr

```

program test_my_chip
interface my_chip_prt {
    input req0 PSAMPLE vca r0;
    output ack0 NDRIVE;
    inout[7:0] dada0 PSAMPLE NDRIVE;
    input req1 PSAMPLE vca r0;
    output ack1 NDRIVE;
    inout[7:0] data1 PSAMPLE NDRIVE;
}

port dt_prt {
    req;
    ack;
    data;
}

bind dt_prt port0 {
    req my_chip_prt.req0;
    ack my_chip_prt.ack0;
    data my_chip_prt.data0;
}

bind dt_prt port1 {
    req my_chip_prt.req1;
    ack my_chip_prt.ack1;
    data my_chip_prt.data1;
}

{
    integer i, j;
    bit dir;
    bit [7:0] my_data;

    extern task handshake with dt_prt(
        bit direction, bit [7:0] h_data
    );

    vca(ON, my_chip_prt);
    for(i = 0; i < 10; i++) {
        my_data = random();
        dir = i;
        handshake with port0(dir, my_data);
    }
}

```

```

hs_tasks.vr
port dt_prt {
    req;
    ack;
    data;
}

extern integer i, j;

local task what_is_ij () {
    printf("top block i = %d j = %d\n", i, j);
}

task handshake with dt_prt (bit direction, bit [7:0] h_data){
    @0,1000 $req == 1'b1;
    $ack = 1'b1;
    @1 $ack <= 1'b0;
    if(direction) $data = h_data;
    else $data == h_data;
    what_is_ij();
}

```

These two files can be compiled separately to generate the object files, *tst_chip.vro*, and *hs_tasks.vro*. A Vera list (.vrl) file lists the object files to be linked at simulation time.

```

tst_chip.vrl
tst_chip.vro
hs_tasks.vro

```

At simulation time, the argument `+vera_mload = tst_chp.vrl` must be called.

18.7 Dynamic Loading of Vera Object Modules

Vera testbenches can be loaded dynamically, without recompiling or reloading the HDL design. This allows for rapid development and easy assembly of regression tests. To enable dynamic loading, Vera provides these user-defined HDL tasks:

```

$vera_load("filename.vro");
$vera_mload("filename.vrl");

```

filename - The *filename* should be an object file when using `$vera_load` or a Vera list file when using `$vera_mload`.

These tasks can be invoked from an interactive prompt or executed from a command file. However, they should not be invoked from within the HDL circuit description files.

Most often, you want to reset the simulation before loading a separate testbench. To reset a simulation, use the `$reset` command:

```
$reset
```

An entire regression can be run by adding the appropriate sequence of resets, loads, and mloads to a command file and executing it.

For example, assume this plus argument is used to launch a simulation:

```
+vera_load=router1.vro
```

Using this plus argument, Vera loads *router1.vro* and runs the testbench `router1`. When the testbench is completed, control passes to the interactive prompt. To dynamically load a second testbench, enter:

```
$reset
$vera_load("router2.vro");
```

This example resets the simulation and runs the testbench `router2`.

The `$vera_load` and `$vera_mload` commands explicitly set the name of the files to be run with the simulation. A reset and restart causes the same testbench to be run again. This allows you to quickly run testbenches again and again. If you want to run a second set of testbenches after the first, you must explicitly set the new files to be run.

Note – If the new testbench has a different interface specification, has different global variables, or declares different HDL tasks, the shell file produced at compile time must change. In this case, you must recompile the source code.

For example, if the third testbench uses several object files contained in a Vera list file, enter:

```
$reset;
$vera_load("")
$vera_mload("router3.vrl");
```

This example resets the simulation. The second line clears the current testbench being used. The third line specifies the list file to be used with the new testbench.

18.8 Running Vera Stand-alone

Vera includes a stand-alone, cycle-based simulator. To invoke the simulator:

```
vera -run vera_executable simulator_options object_files
```

vera_executable - The *vera_executable* is the compiled Vera source code being run.

simulator_options - The simulator options and their definitions are listed in Table 18-6.

Table 18-6 Simulator Options

Option	Definition
-e	Terminates simulation on error
-f	Specifies list of Vera object files to load
-s	Pauses simulation on error
-t	Opens debugger in text mode

object_files - Object files used as part of the simulation are listed in the command line, separated by white space.

-e

The **-e** option terminates the simulation when an error is encountered. The syntax is:

-e

-f

The **-f** option loads multiple object files from a Vera list file. The syntax is:

-f filename

filename - The *filename* is the name of the .vrl file containing the list of object files to be run.

-s

The **-s** option pauses the simulation when an error is encountered. The syntax is:

-s

When an error is encountered, the simulation pauses until you enter a carriage return.

-t

The **-t** option launches the text debugger when an error is encountered. The syntax is:

-t

The default setting launches the graphical debugger when an error is encountered.

Vera stand-alone is a cycle-based simulator, which runs without an HDL description and in conjunction with C for high-level modeling. All interface clocks are driven with the same timing. These system functions and tasks work differently:

delay() returns immediately (with a warning).

get_time() always returns zero.

Note – To use the Vera stand-alone cycle-based simulator, you must compile an executable. This procedure is documented in Section 2.6, “Running the Vera Stand-Alone Simulator.”

1. The first step is to create a new project in the Vera IDE. This is done by clicking on the 'File' menu and selecting 'New Project...'. A dialog box will appear where you can specify the project name and location. Once you have created the project, you can add a new component to it by clicking on the 'Component' menu and selecting 'Add Component...'. This will open a list of available components, and you can select the one you want to add. Once you have added the component, you can start configuring it by clicking on the 'Configuration' menu and selecting 'Configure Component...'. This will open a configuration window where you can set various options for the component. Finally, you can compile the project by clicking on the 'Build' menu and selecting 'Build Project...'. This will compile the project and generate the output files.

19. Vera Source-Level Debugger

This chapter details the Vera Source-Level Debugger. It includes these sections:

- Overview
- Interactive Debugging
- Contexts
- Debugger Expressions
- Text-Based Debugger Commands
- Graphical Debugger

19.1 Overview

The Vera debugger is an interactive interface that allows you to step through Vera code, granting tremendous debugging power. It includes an easy-to-use graphical front end along with a powerful text-based interface.

Vera incorporates powerful simulation debugging tools within its debugger, including the notion of cycles and timing, interface clocks, multiple thread analysis, and support for re-entrant and recursive code. These tools allow you to control and observe the simulation from the testbench perspective.

The Vera debugger is used to debug Vera code, and it does not affect the debugging or execution of other HDL code or waveform display tools. Thus, you can use the Vera source-level debugger in conjunction with existing HDL graphic packages, HDL debuggers, and waveform displays.

19.2 Interactive Debugging

The Vera debugger is activated when one of several events occur:

- A Vera breakpoint is encountered
- The Vera debugger is invoked from the HDL command line
- A verification error such as a failed expect or a timeout occurs
- A runtime error such as an improper bind or invalid index occurs

If you want to use the graphical debugger while running a simulation with an HDL, you must run the simulation with the `+vera_debug_on_error` plus argument for Verilog (see Section 18.5, "Running Vera with HDLs") or the `vera.ini` token `vera_debug_on_error on` for VHDL (see Section 17.3.2, "VHDL Plus Arguments").

You should compile all Vera source files with the `-g` switch if you want to run the Vera debugger with all symbolic information for the files included.

19.3 Contexts

In Vera, a context refers to a thread of execution. Vera assigns each context a unique ID internally, which is used to identify contexts within the debugger. When using the debugger, you can switch between contexts to examine code execution in any context within the simulation.

A new context is generated under several circumstances:

- A Vera simulation is launched
- A fork is executed
- A function or task is called

Note that because Vera allows non-blocking forks, a procedure may finish, but its context may still be alive (as children it has forked may not have finished yet). This is the reason for assigning a context not only to each fork, but also to each procedure call.

The context determines the variables that are visible in the debugger. The variables in the current context are always visible in the debugger. This means that any local variables as well as global variables are visible. If shadow variables are used within a fork context, they are visible and hide variables of the same name in the parent context.

19.4 Debugger Expressions

The Vera debugger supports a subset of the valid Vera expressions. Expressions can contain variables, interface signals, and integer constants. Expressions can also contain these operators: -, *, /, %, &, !, ^, &~, !~, ^~, &&, ||, <<, >>, <, >, <=, >=, =, !=, ==, !=, =?=. Expressions can also contain these unary operators: -, ~, !, &, !, ~&, ~!, and ~^.

Expressions can use variables. However, the expression can only be evaluated if the variable is visible in the current context.

19.5 Text-Based Debugger Commands

When the text-based debugger is launched, you can enter commands at the debugger prompt. A command consists of a single line of input, starting with a command name. Multiple commands can be issued using semicolons (;) to separate them. If there is a conflict between an identifier and a Vera debugger command, preface the identifier with a slash (/). For example:

```
print \print
```

The valid text-based debugger commands are listed with a brief definition in Table 19-1.

Table 19-1 Vera Text-Based Debugger Commands

Command	Definition
.	Continue
,	Step
..	Step in context
..	Next in context
break line_num	Set break point
clear break line_num	Clear break point
show break	Show break points
print exp	Prints value of expression
assign var=expr	Assigns value to expression
show vars	Shows accessible variables
show context	Shows active context lists
source file filename	Reads commands from a file
end	Terminates a source file
echo "string"	Echoes a string
context context_id	Switches contexts
up	Changes to parent context
down	Changes to child context
where	Shows current location
list line_num,#lines	Lists source of context
show file	Shows source files
watch add "exp"	Adds a watch expression
watch change watch_id "exp"	Changes a watch expression
watch insert watch_id "exp"	Inserts a watch expression
watch remove watch_id	Removes a watch expression
show watch	Prints the watch list
save filename	Saves environment settings
restore filename	Restores environment settings
quit	Exits the simulator
close	Closes graphical debugger
help	Prints debugger commands
radix num_base	Sets the default radix
window	Launches graphical debugger

19.5.1 Execution Control

The Vera debugger has several commands used for execution control.

The `.` command is used to continue the Vera simulation. The simulation continues from the last break-point or error point.

The `,` command is used to step through the Vera program. The very next executable statement in the Vera program is executed, regardless of context, and the simulation stops.

The `..` command is used to execute the next step within the current context. When this command is invoked, the very next Vera statement within the current context is executed, and the simulation stops. If other code must be executed before the next statement in the context is executed, that code is executed. This command executes the next step in the context unless that step is within a fork/join none. In that case, the first line after the fork/join none is executed.

The `„` command is used to execute the next statement within the current context. Statements within sub-routines and forks are not executed with this command. Instead, the first statement after a subroutine or fork is executed.

break

The `break` command sets a break point at the specified line within the specified file. The syntax is:

```
break filename@ line_number
```

filename - The *filename* is the filename in which the break point is to be set. If no filename is set, the current file is used.

line_number - The *line_number* specifies the line where the break point is set.

This is an example of a `break` command:

```
break myfile@ 53
```

clear break

The `clear break` command clears the specified break point. The syntax is:

```
clear break filename@ line_number
```

filename - The *filename* is the filename from which the break point is to be cleared. If no filename is set, the current file is used.

line_number - The *line_number* specifies the line where the break point to be cleared is. If no line number is specified, all breaks in the specified file are cleared.

This is an example of a **clear break** command:

```
clear break myfile@ 53
```

show break

The **show break** command lists the active break points in all files of the simulation.

19.5.2 Displaying and Updating Data

The Vera debugger has several commands used to display and update data.

print

The **print** command evaluates the specified expression and shows the result. The syntax is:

```
print expression base
```

expression - The *expression* can be any valid debugger expression, including arrays.

base - The *base* determines the numeric base of the expression. It must be 0 (auto), 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). The default is set with the **radix** command.

This is an example of the **print** command:

```
print i+100
```

If the expression is an array, you can optionally specify the number of array elements to print:

```
print array_name num_elements
```

If you do not specify the number of elements to print, the current array limit is printed.

limit array

The **limit array** command sets the number of array elements printed with the **print** command. The syntax is:

```
limit array num_elements
```

The default value is 256. If you do not specify a number, the command returns the current array limit.

assign

The **assign** command assigns a value to the specified variable. The syntax is:

```
assign variable = expression
```

variable - The *variable* can be any variable visible in the current context.

expression - The *expression* can be any valid debugger expression.

Note – Assignments to interface signals are done immediately, without waiting for the corresponding drive clock edge.

show vars

The `show vars` command shows the variables in the current context. It displays the data showing the hierarchy of the call/block structure. For example:

```
show vars
List of active variable at program main 162 in ..
File: /foo/foo.vr

program main
{
    integer i;
    integer j;

    class class_b
    {
        integer int_array[50];
        integer int_asoc[];
        task local_array_check
        (
            integer flag,
            integer flag1
        )
    {
        integer local_var;
```

19.5.3 Accessing Context Information

The Vera debugger supports several command used to access context information.

show context

The `show context` command shows the active contexts. For example:

```
show context
Context_ID:406 STATUS:READY
    Location: CALL in program main (cls_dbg.vr, line 162);
    READY in task foo (bar.vr, line 77)
Context_ID:0 STATUS:CALLING
    Location: CALL in program main (cls_dbg.vr, line 162)
```


source file

The source file command executes commands from a specified file. The syntax is:

```
source file filename
```

filename - The *filename* is the name of the file that contains the commands.

The source file command is particularly useful when the same commands are issued repeatedly.

end

The end command is used to terminate the execution of a source file.

echo

The echo command is used to echo a string to the terminal window. The syntax is:

```
echo "string"
```

string - The *string* can be any string of characters, encapsulated by quotes.

context

The context command is used to change contexts. The syntax is:

```
context context_id
```

context_id - The *context_id* is the unique numeric identifier assigned to a context. If no *context_id* is specified, the command returns the identifier of the current context.

up

The up command changes contexts to the parent context. The parent context is the context from which the current context was called or forked.

down

The down command changes contexts to the child context. If there is more than one child context, the command returns a message and the context is not changed.

where

The where command shows the last executed Vera statement and context. For example:

```
where
Location: CALL in program main (foo.vr, line 162);
READY in task bar (foo.vr, line 77)
```

19.5.4 Accessing Source Files

The Vera debugger provides several commands for accessing source files.

list

The **list** command lists the specified lines of Vera source code. The syntax is:

```
list filename@ line_number, num_of_lines
```

filename - The *filename* is the filename from which the source code is taken. If no *filename* is specified, the code from the current file is displayed.

line_number - The *line_number* specifies which line of code is the first line displayed. If no *line_number* is specified, the list of source code starts five lines before the current line.

num_of_lines - The *num_of_lines* specifies the number of lines displayed. If it is not specified, 20 lines of code are displayed.

In the listing, there is a column between the line number and the source code. In that column, an S indicates a break point, and a > indicates the current line. For example:

```
list , 7
157      trigger(OFF, clsa.which_event(i));
158      }
159
160      i = 0;
161      j = 0;
162 >  foo(0, 2);
163
164 S   i++;
165      printf("i = %0d\n", i);
```

Note that blank lines are numbered but do not count towards the number of lines of code displayed.

show files

The **show files** command shows the names of the loaded Vera source files.

19.5.5 Watch Commands

The Vera debugger supports several commands used in conjunction with its watch windows. The watch windows are used to show all the variables in the current context, including global variables, and their values throughout the simulation. A second set of watch windows shows user-selected expressions and their values throughout the simulation. Each expression in the watch windows is given a unique numeric identifier used to manipulate watch expressions.

watch add

The **watch add** command adds an expression to the watch windows. The syntax is:

```
watch add "expression" base
```

expression - The *expression* is any valid debugger expression using variables in the simulation. Only when the context of the variables is shown will the expression be evaluated.

base - The *base* must be 0 (auto), 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). The default setting is set with the **radix** command.

watch change

The **watch change** command changes an existing watch expression. The syntax is:

```
watch change watch_number "expression" base
```

watch_number - The *watch_number* is the unique numeric identifier of the watch expression you want to change.

expression - The *expression* is the new expression you want to assign to the watch number. It can be any valid debugger expression. If the *expression* is omitted and the *base* is included, only the *base* changes.

base - The *base* must be 0 (auto), 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). The default setting is 0. If it is omitted, the base from the previous expression is used.

watch insert

The **watch insert** command inserts a watch expression with a specified watch number. The syntax is:

```
watch insert watch_number "expression" base
```

watch_number - The *watch_number* is the unique numeric identifier of the watch expression you want to insert.

expression - The *expression* is the new expression you want to assign to the watch number. It can be any valid debugger expression. If the *expression* is omitted and the *base* is included, only the *base* is inserted.

base - The *base* must be 0 (auto), 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). The default setting is 0. The default is set with the **radix** command.

watch remove

The **remove watch** command removes a watch expression. The syntax is:

```
remove watch watch_number
```

watch_number - The *watch_number* is the number of the watch expression you want to remove.

If you do not specify a watch number, all of the watch expressions are removed.

show watch

The **show watch** command shows all the current watch expressions and watch number identifiers.

19.5.6 Debugging Environment Control.

The Vera debugger supports several commands for controlling the debugging environment.

save

The **save** command saves the current environment settings to a file. The syntax is:

```
save filename
```

filename - The *filename* specifies which file the environment settings are saved to. If no filename is specified, the settings are saved to *vera_debug_auto.rc*.

The environment settings that are saved include: breakpoints, the current radix, array limits, and watch expressions.

restore

The **restore** command restores environment settings from a specified file. The syntax is:

```
restore filename
```

filename - The *filename* specifies the file containing the environment settings to be restored. If no filename is specified, the settings are restored from *vera_debug_auto.rc*.

quit

The **quit** command exits the simulation. When you quit, the current environment settings are saved in the file *vera_debug_auto.rc*.

19.5.7 Miscellaneous Commands

The Vera debugger supports a set of miscellaneous debugger commands.

close

The **close** command closes the graphical debugger interface.

help

The **help** command lists the debugger commands with a short definition of each.

radix

The **radix** command sets the default radix. The syntax is:

```
radix base
```

base - The *base* must be 0 (auto), 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). The default setting is 0. The default is 0.

window

The `window` command launches the graphical debugger interface.

19.6 Graphical Debugger

The Vera graphical debugger can be invoked from the text-based debugger or launched when the debug mode is entered if the `-t` runtime option is not used.

The button names and their corresponding text-based equivalent are listed in Table 19-2.

Table 19-2 Graphical Debugger Button Definitions

Button Name	Text-based Equivalent
<code>show vars</code>	<code>show vars</code>
<code>print <exp></code>	<code>print</code>
<code>watch <exp></code>	<code>watch add</code>
<code>unwatch <exp></code>	<code>watch remove</code>
<code>close</code>	<code>close</code>
<code>quit</code>	<code>quit</code>
<code>where</code>	<code>where</code>
<code>step</code>	<code>,</code>
<code>stepc</code>	<code>..</code>
<code>nextc</code>	<code>..</code>
<code>cont</code>	<code>.</code>
<code>up</code>	<code>up</code>
<code>down</code>	<code>down</code>
<code>brk <sel></code>	<code>break</code>
<code>clr brk <sel></code>	<code>clear break</code>
<code>auto</code>	<code>radix 0</code>
<code>set radix</code>	<code>radix</code>

When using the `print`, `watch`, and `unwatch` command buttons, you must use the mouse to highlight the expression you want to print, watch, or unwatch.

When using the `brk` and `clr brk` command buttons, you must use the mouse to highlight the line you want to assign a break to or clear a break from.

19.6.1 Usage Notes

The current simulation cycle and simulation time are shown in the top bar of the debugger window. If you are running multiple modules, the module name is also shown in the bar.

The watch windows to the right of the debugger interface show the watch expressions and variables with their values throughout the simulation. These expressions are only evaluated when the variables they include are in the current context (see Section 19.5.5, "Watch Commands").

Right-clicking on an expression, array, or class in a watch window invokes an expansion window that shows an expanded value of the chosen element. Expressions are expanded and shown in their entirety when they are selected. When an array is selected, the elements of that array are shown (the number of elements shown depends on the array limit). When a class is selected, the class is expanded and shown in the expansion window. Further, arrays and classes within expansion windows can be expanded (single values cannot be expanded from an expansion window).

Note – Expansion windows are static and only show the selected element at that moment of time in the simulation. When you step through the program, they are cleared.

The bar under the context window always shows the current context. Double-clicking any context in the list changes the current context. The bar under the file window always shows the current file. Double-clicking any file in the list changes the current file. When the current context and current file are separate files, the color of their respective bars does not match.

Text-based debugger commands can be issued from the command box at the upper right corner of the debugger window.

19.6.2 Notes for Tcl/Tk Users

By default Vera uses a copy of the Tcl/Tk wish shell and libraries, which reside in the Vera directory. You need to have Tcl 7.5 and Tk 4.1(or higher) installed and within your path. Note that Tcl/Tk is not part of Vera. However, Vera 3.1 and higher includes a pre-compiled version of Tcl/Tk, which gets picked up automatically by the source debugger.

If you want to extend or modify the graphic window, you can edit `$VERA_HOME/bin/vera_debugger.tcl`, create a new version, and use it by specifying your script instead of the Vera default, from the debugger prompt:

```
window wish_name tcl_file
```

If you modify this script, Vera invokes it as follows:

```
wish -f $VERA_HOME/bin/vera_debugger.tcl process_id
```

Appendix A. Quick Reference

This appendix includes quick reference information in these areas:

- Vera Files
- Vera System Functions and Tasks
- Compiler Switches
- Vera Debugger Commands

A.1 Vera Files

These are the Vera file extensions and their definitions.

Filename	Definition
filename.vr	Vera source file
filename.vro	Compiled Vera object file
filename.vrh	Vera header file
filename.if.vrh	Vera interface file
filename.vrl	Vera list file
filename.vshell	Vera shell file
filename.test_top.v	Vera Test-top file for Verilog
filename.top	Vera top file for VHDL
filename_top.vhd	Vera test-top file for VHDL
filename_intf.vhd	VHDL interface file
vera.ini	Vera initialization file for VHDL
filename.vcon	Configuration file
filename.proj	Project file
filename.shell.v	Vera shell file for multiple modules

A.2 Vera System Functions and Tasks

```

alloc(type, type_id, type_count [, key_count]);
assoc_index (command, array_name, index);
atobin();
atohex();
atoi();
atooct();

```

```

backref(index);
bittostr(bit[high:low] bit_string);
boundary(which);
cast_assign(dest_var, source_exp [, check]);
constraint_mode(switch [, constraint_name]);
coverage(command [, object_list] [, "filename"]);
delay(time);
error("fmt_str");
error_mode(type, error_class);
exit(status);
fclose (file_descriptor);
flag([value]);
fopen ("filename", "type");
fprintf (file_descriptor, fmt_str);
freadb(file_descriptor);
freadh(file_descriptor);
freadstr(file_descriptor, mode);
get_bind([ID]);
get_bind_id(bind_expression);
getc(i);
get_cycle();
get_plus_arg(request, plus_arg);
get_status();
get_status_msg();
get_systime();
get_time(word);
itoa(i);
len();
mailbox_get(wait_option, mailbox_id [, dest_var [, check_option] ]);
mailbox_put(mailbox_id, data);
mailbox_receive(wait_option, mailbox_id);
mailbox_send(mailbox_id, data);
match(pattern);
pack(array, index, left, right);
postmatch();
prematch();
printf("fmt_str", arg1, arg2, ..., argN);
prodget([production_name [, occurrence_number] ]);
prodset (value [, production_name [, occurrence_number] ]);
putc(i, "char");
query(command [ [ [, bin_type], bin_pattern], operand, hit]);
rand48([seed]);
rand_mode(switch [, variable_name [, index] ]);
random([seed]);

```



```

randomize();
region_enter(wait_option, region_id, value1, value2, ..., valueN);
region_exit(region_id, value1, value2, ..., valueN);
rewind(file_descriptor);
search(pattern);
semaphore_get(wait_option, semaphore_id, key_weight);
semaphore_put(semaphore_id, key_weight);
signal_connect(port_signal, target_signal [, attributes [, clock] ]);
sprintf(string_name, string_format);
sscanf(string_name, string_format);
stop();
substr(i, j);
suspend_thread();
sync(sync_type, event1, event2, ... eventN);
thismatch();
timeout(object_type, cycle_limit [, object_id]);
trace(request, object [, id_level]);
trigger([trigger_type,] event_name);
unpack(array, index, left, right);
urand48([seed]);
urandom([seed]);
vca(switch [, signal_name]);
vera_free_arg(vera_data);
vera_get_arg(index);
vera_num_arg();
vera_put_arg(index, vera_data);
vera_return_value(vera_data);
vsv_call_func(connection, time_mode, func_name, return_value [, opt_args]);
vsv_call_task(connection, time_mode, task_name [, opt_args]);
vsv_close_conn(connection);
vsv_get_conn_err();
vsv_make_client("host", port, authentication);
vsv_make_server(port, authentication [, verbose]);
vsv_up_connections(timeout);
vsv_wait_for_done();
vsv_wait_for_input(wait_mode);
wait_child();
wait_var(variables);

```

A.3 Compiler Switches

These are the compiler switches for use with the Vera simulator.

Option	Definition
-help	Lists valid compiler options
-cmp	Compiles Vera source files
-alim	Sets maximum number of elements in arrays
-ansi	Invokes ansi preprocessor
-g	Invokes compiler using debug information
-h	Updates .vrh file
-hnu	Updates .vrh file only if it has changed
-i	Produces ASCII interface definition
-ip	Compiles an IP core
-max_error	Sets the maximum number of errors before compilation failure
-q	Compiles in quiet mode
-timescale	Sets timescale
-top	Creates VHDL top file
-vhd1	Processes VHDL top file
-D	Specifies text macro
-I	Includes directory in search path
-pp	Invokes preprocessor
-proj	Creates Vera shell files from project file
-run	Runs simulation
-tem	Invokes Vera template generator
-c	Specifies clock signal
-d	Specifies dump file
-nav	Uses Magellan for simulation control
-t	Specifies Verilog module name
-v	Returns the compiler version number
-vcon	Generates a .vcon file

These are the switches for use with Vera-CS.

Option	Definition
-e	Exits simulation on error
-f	Includes .vrl file in the compile
-i	Specifies initialization file
-s	Pauses simulation on error
-t	Invokes text-based debugger

These are the Vera plus arguments.

Plus Argument	Definition
+vera_load	Specifies a single object file
+vera_mload	Specifies a file with a list of object files
+vera_stop_on_error	Execution stops on simulation error
+vera_continue_on_error	Execution continues after verification error
+vera_exit_on_error	Exits simulator on error
+vera_finish_on_end	Exits simulator when simulation finishes
+vera_debug_on_error	Invokes Vera debugger on error
+vera_text_debug	Launches the Vera text debugger on error
+vera_semaphore_size	Specifies maximum number of semaphores
+vera_region_size	Specifies maximum number of regions
+vera_mailbox_size	Specifies maximum number of mailboxes

A.4 Vera Debugger Commands

These are the Vera text-based debugger commands.

Command	Definition
.	Continue
,	Step
..	Step in context
..	Next in context
break <i>line_num</i>	Set break point
clear break <i>line_num</i>	Clear break point
show break	Show break points
print <i>exp</i>	Prints value of expression
assign <i>var=expr</i>	Assigns value to expression
show vars	Shows accessible variables
show context	Shows active context lists
source <i>file filename</i>	Reads commands from a file
end	Terminates a source file
echo "string"	Echoes a string
context <i>context_id</i>	Switches contexts
up	Changes to parent context
down	Changes to child context
where	Shows current location
list <i>line_num,#lines</i>	Lists source of context
show file	Shows source files
watch add "exp"	Adds a watch expression
watch change <i>watch_id</i> "exp"	Changes a watch expression
watch insert <i>watch_id</i> "exp"	Inserts a watch expression
watch remove <i>watch_id</i>	Removes a watch expression
show watch	Prints the watch list
save <i>filename</i>	Saves environment settings
restore <i>filename</i>	Restores environment settings
quit	Exits the simulator
close	Closes graphical debugger
help	Prints debugger commands
radix <i>num_base</i>	Sets the default radix
window	Launches graphical debugger

These are the graphical bugger button definitions.

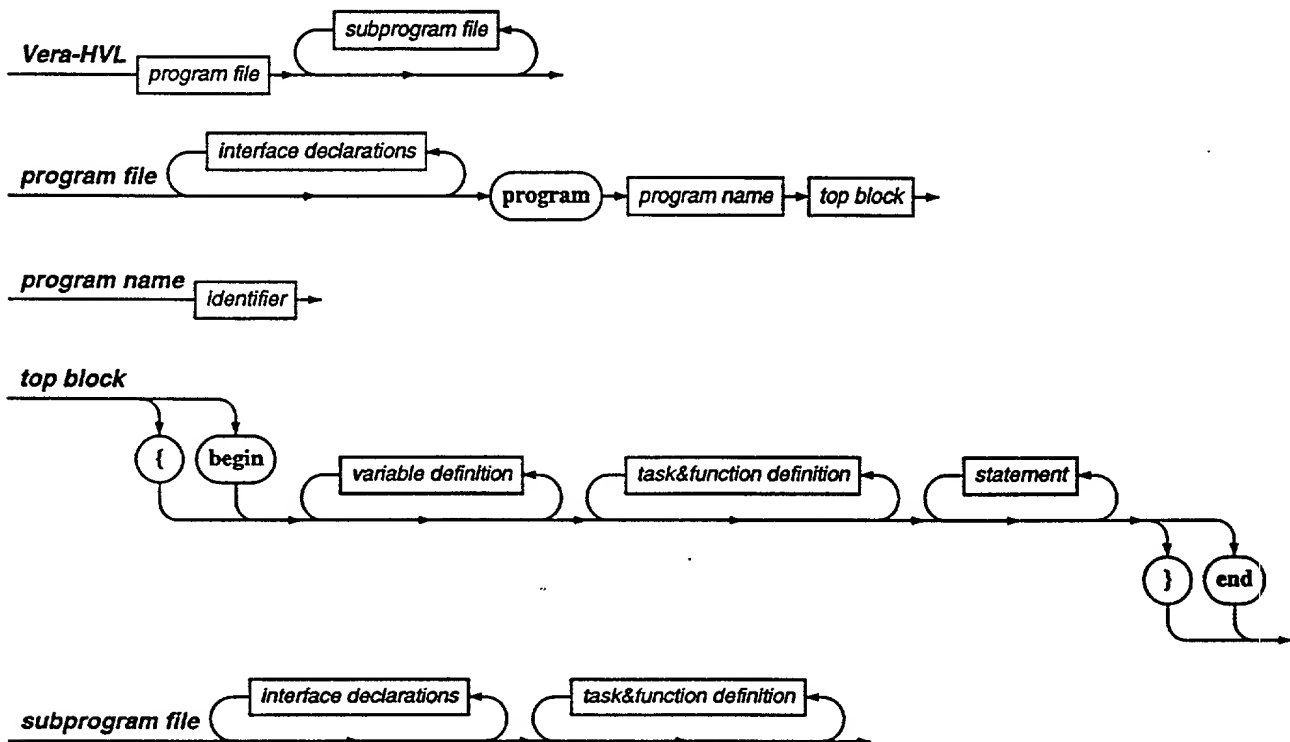
Button Name	Text-based Equivalent
show vars	show vars
print <exp>	print
watch <exp>	watch add
unwatch <exp>	watch remove
close	close
quit	quit
where	where
step	,
stepc	..
nextc	..
cont	.
up	up
down	down
brk <sel>	break
clr brk <sel>	clear break
auto	radix 0
set radix	radix

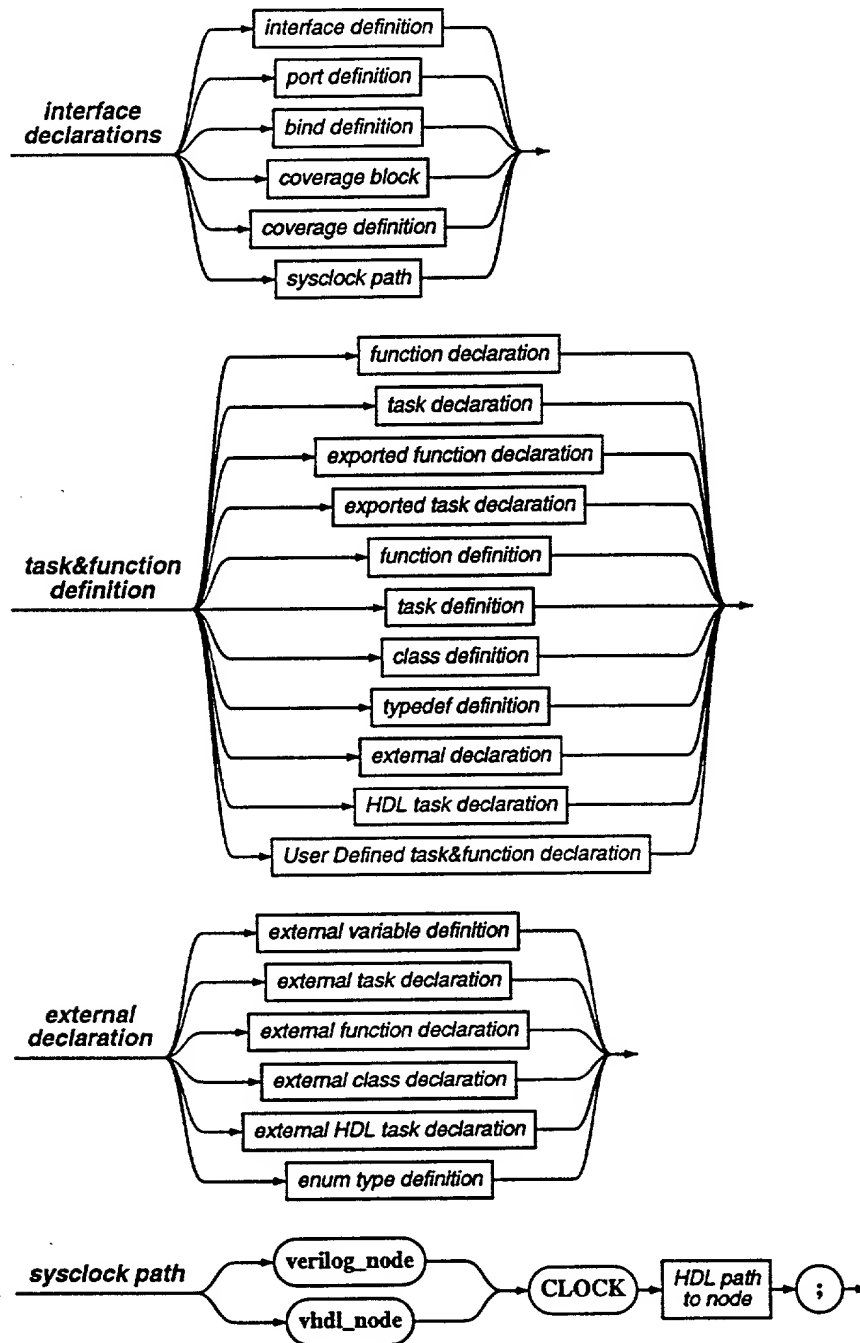
Copyright © 2000 Synopsys, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or by any information storage and retrieval system, without the prior written permission of Synopsys, Inc.

Appendix B. Vera 4.0 BNF Diagrams

B1. Vera-HVL: The Language

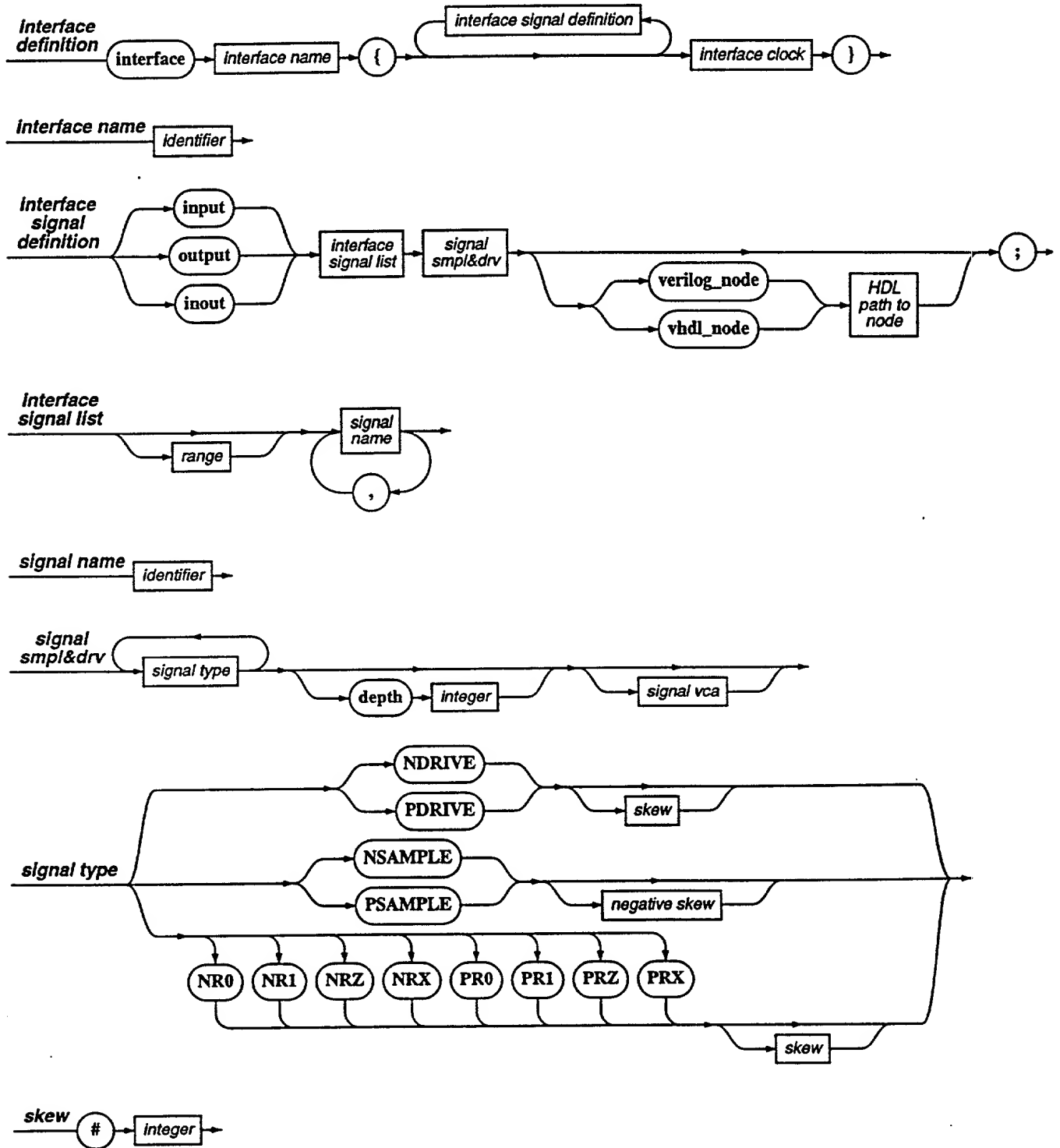
B1.1 Program Structure



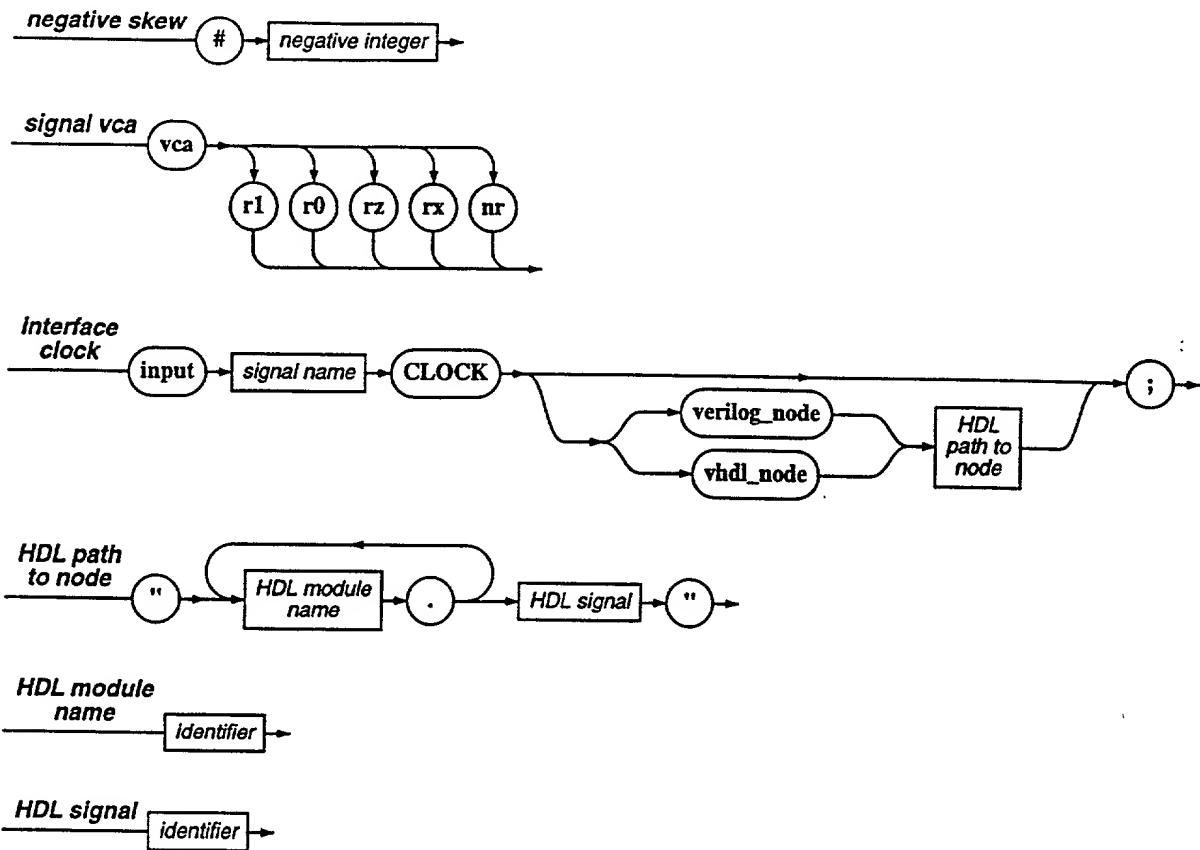


(375)

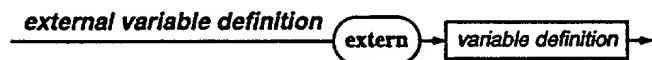
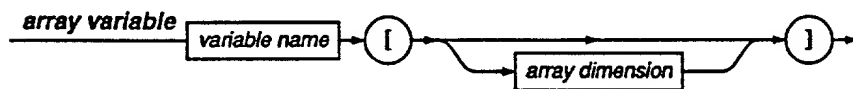
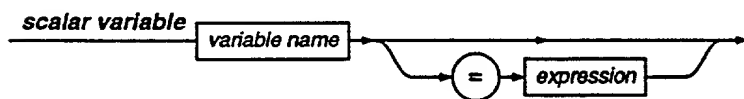
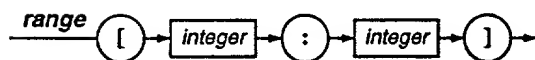
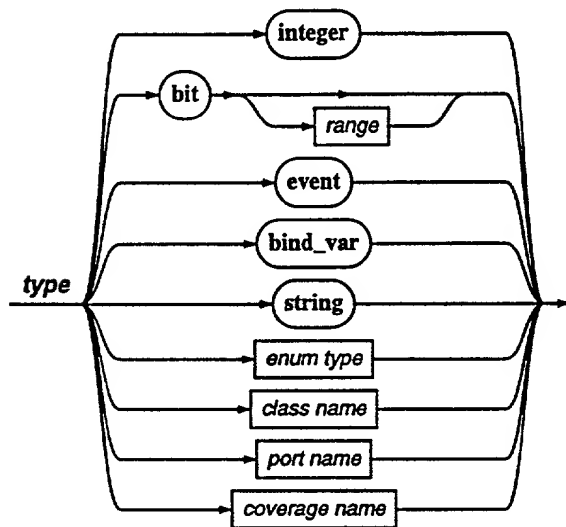
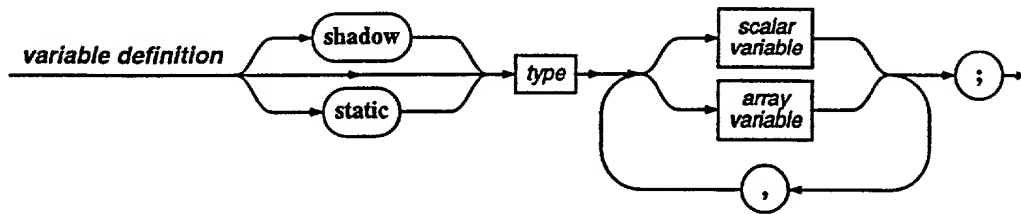
B1.2 Interface Definition



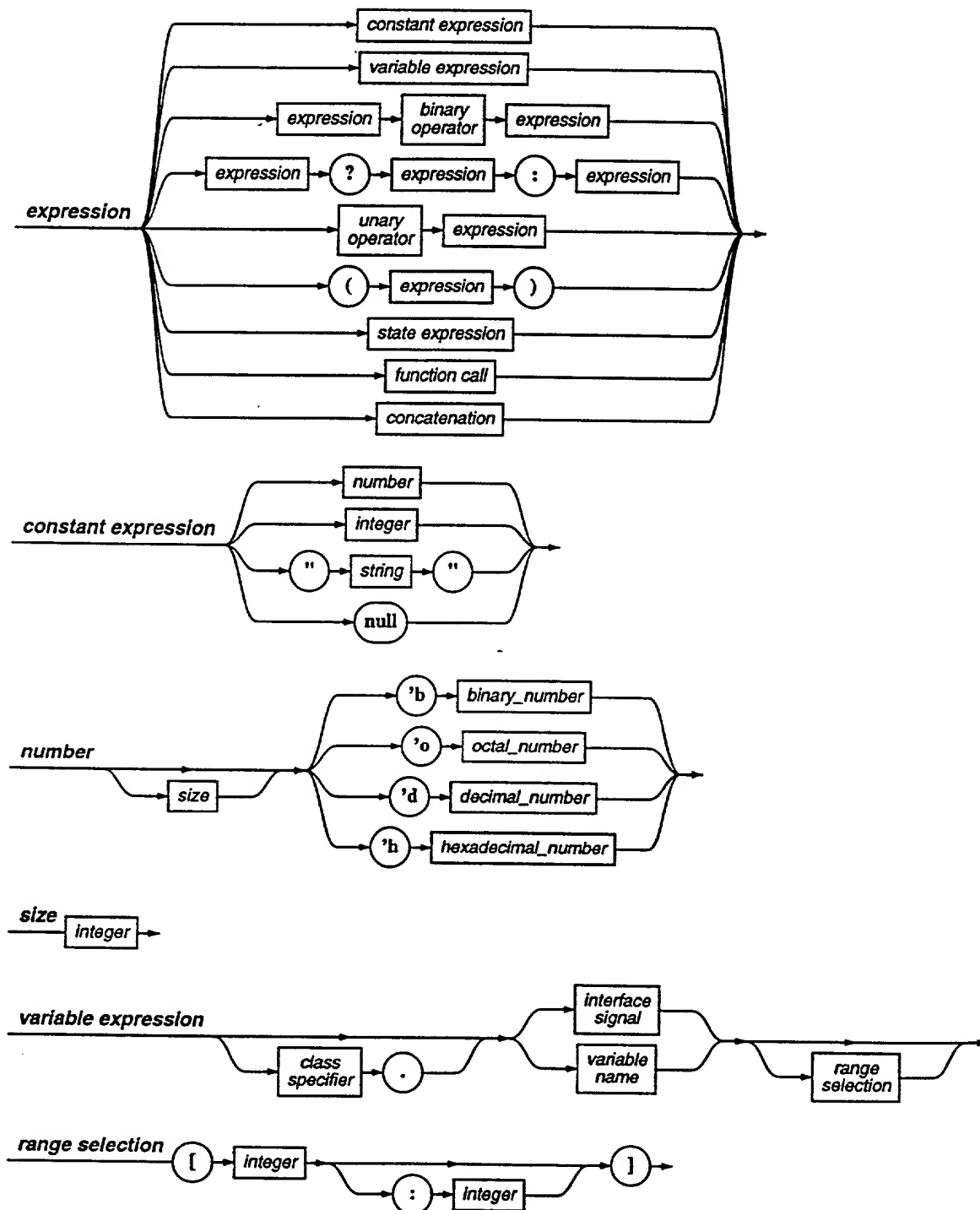
(376)

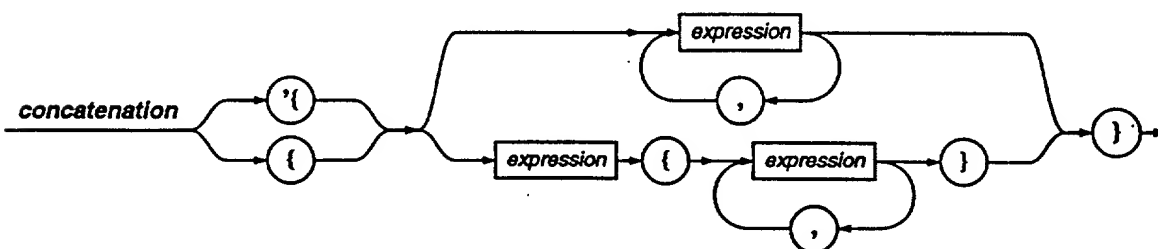
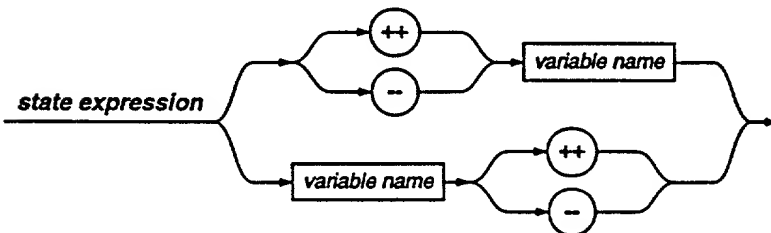
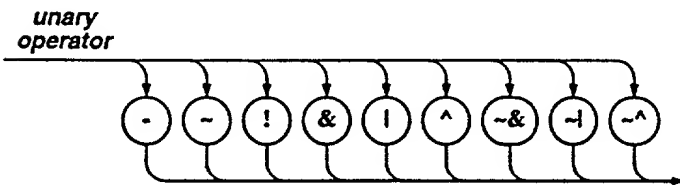
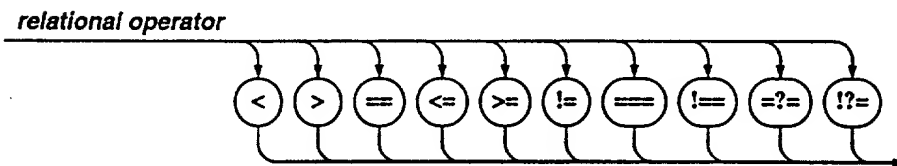
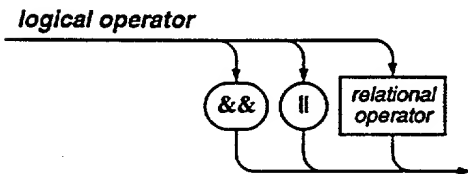
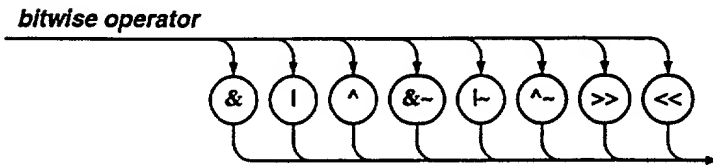
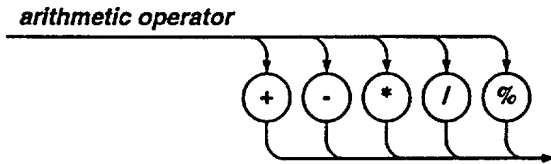
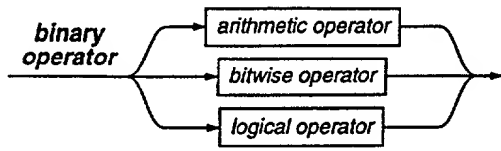


B1.3 Variable Definition

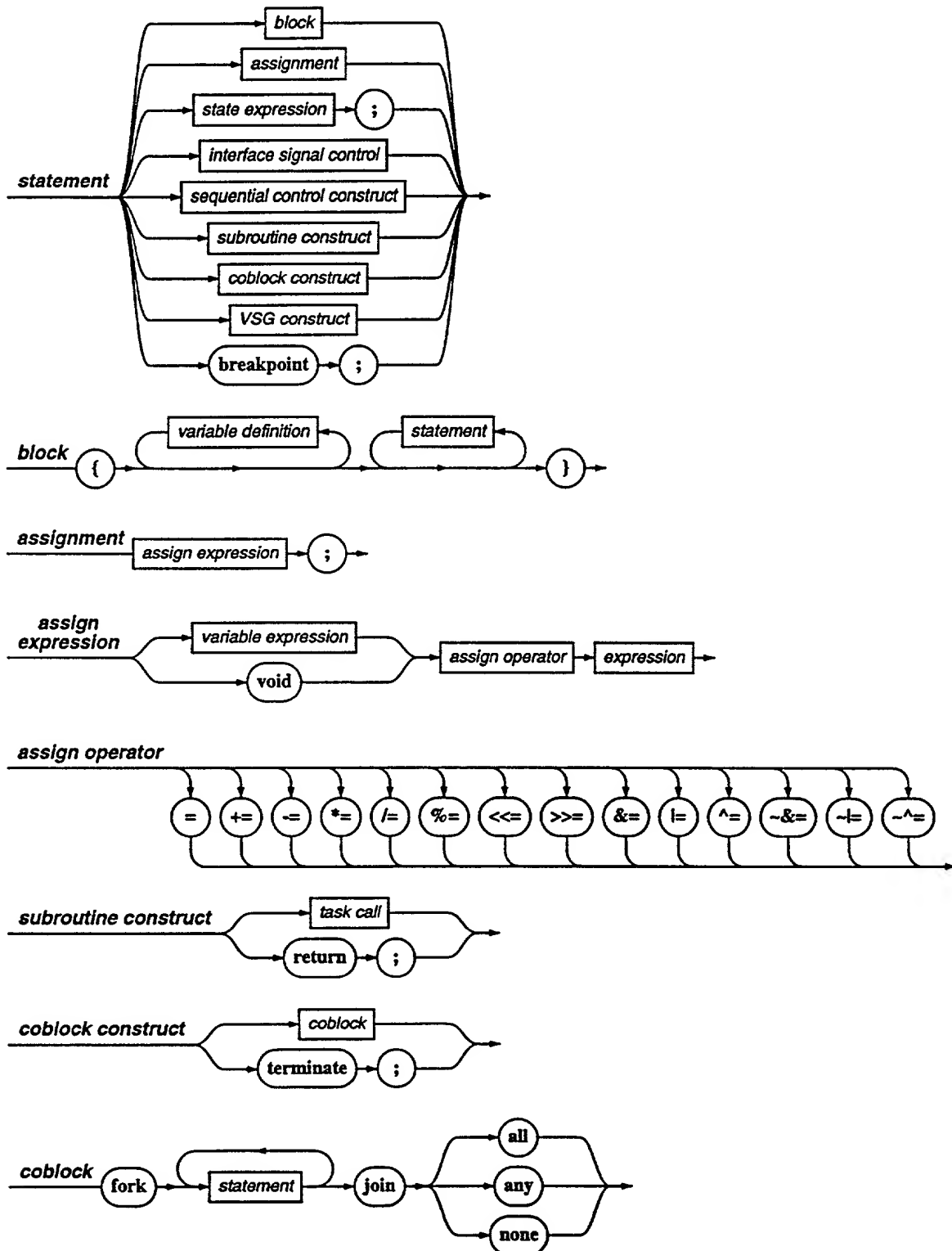


B1.4 Expressions

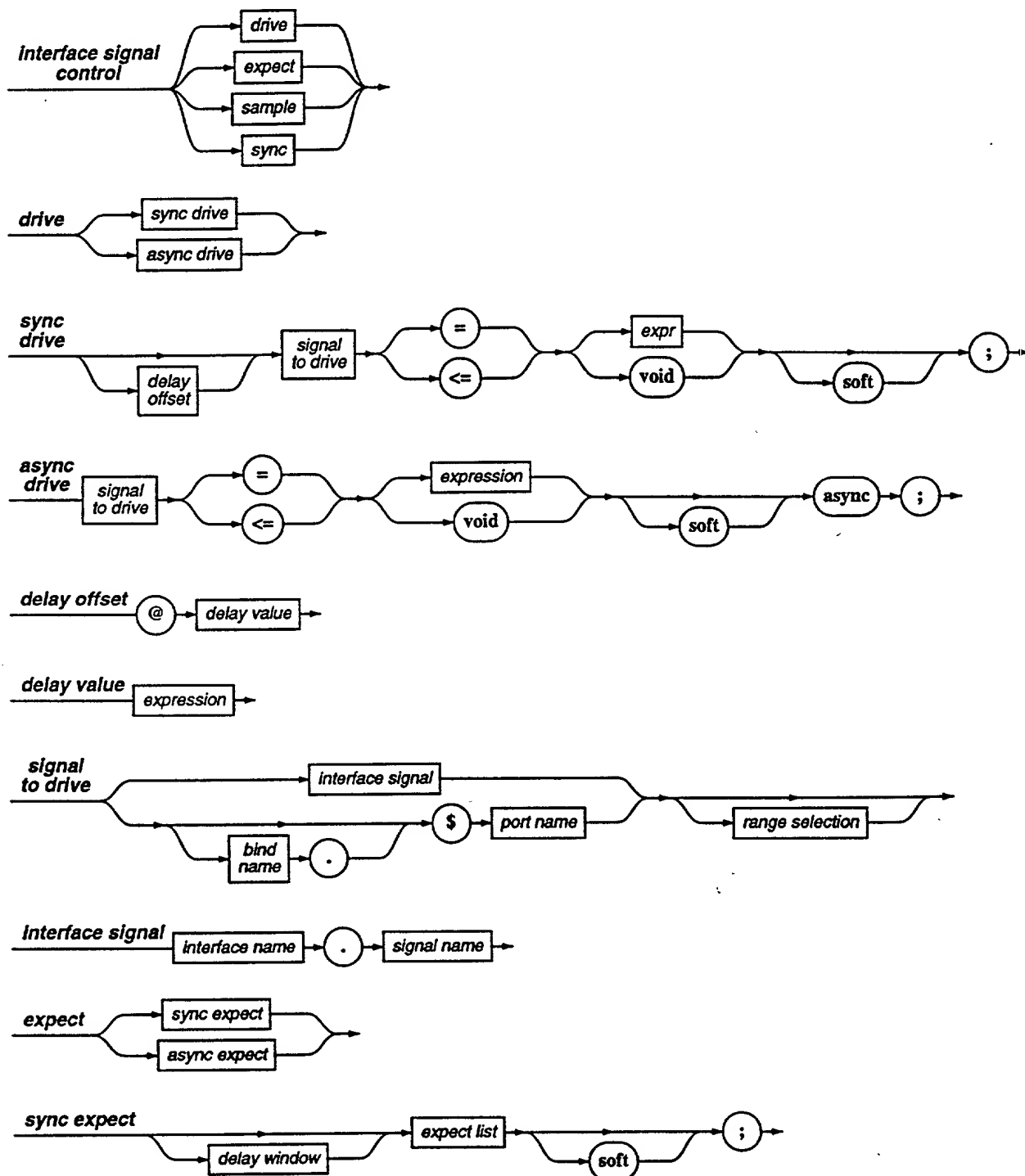


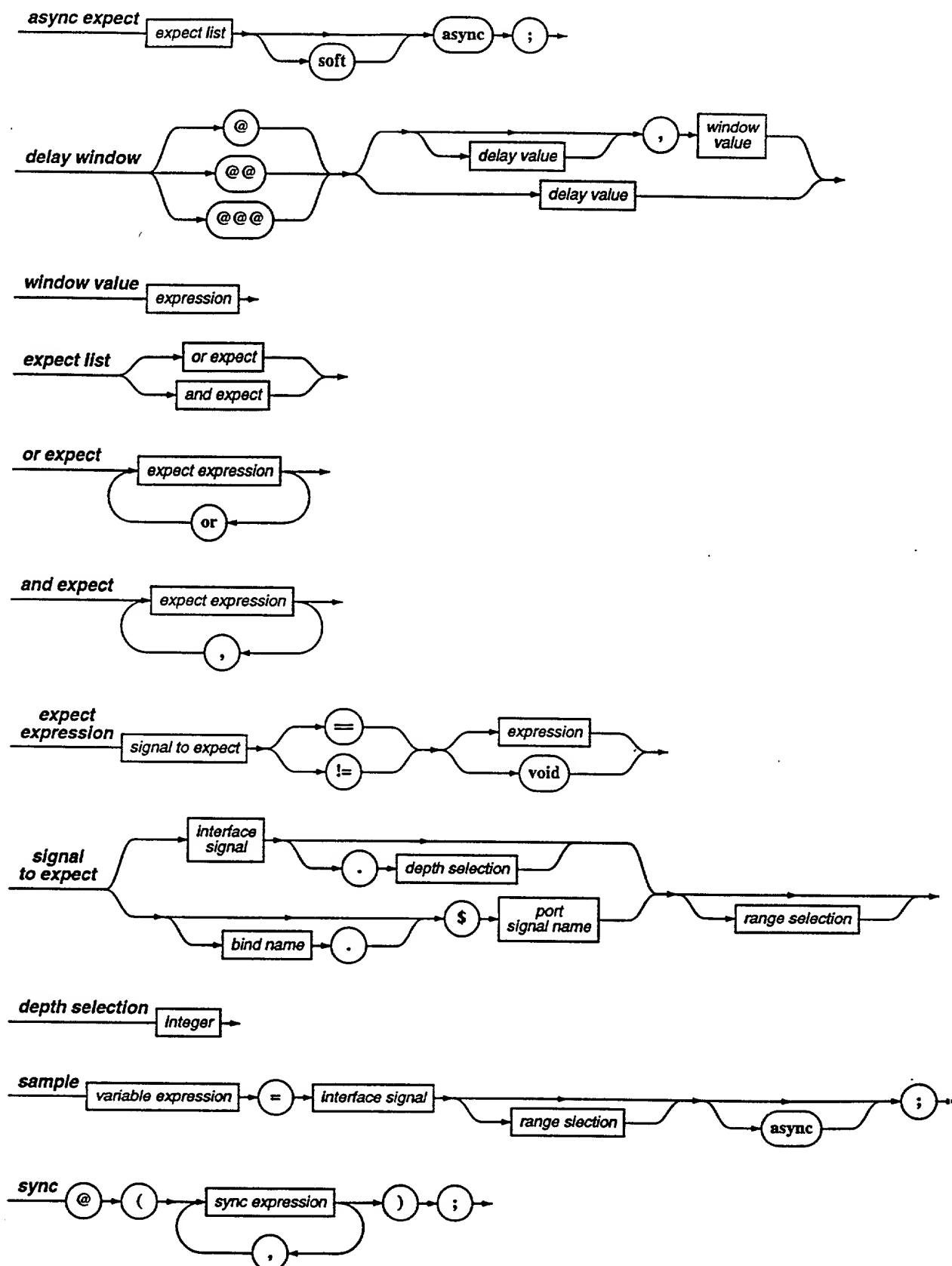


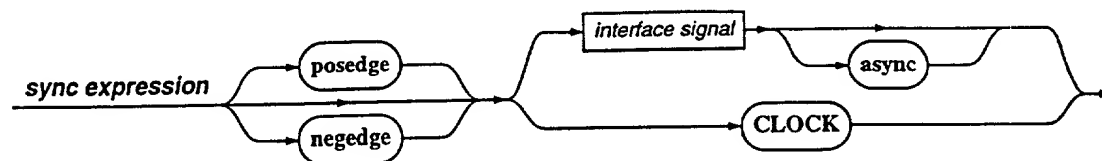
B1.5 Statements



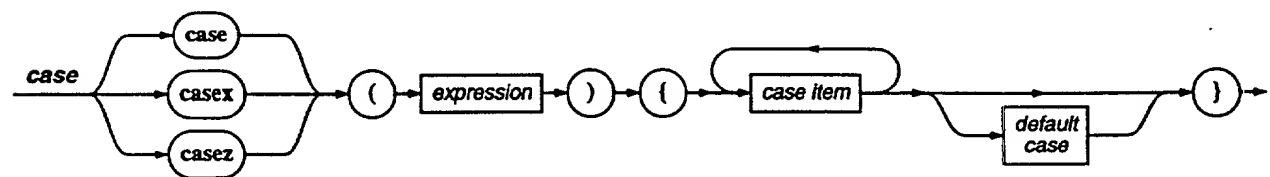
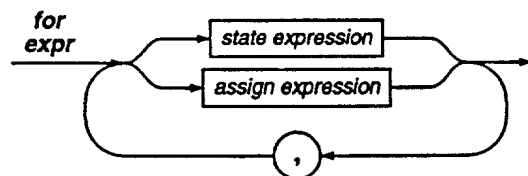
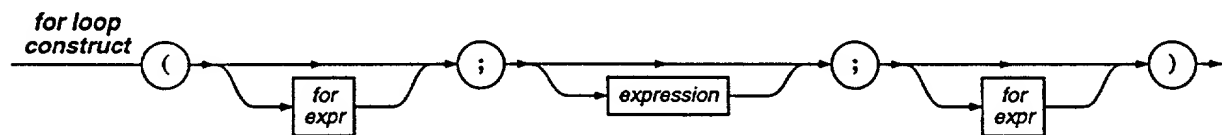
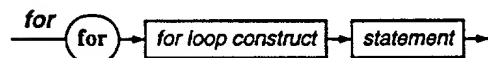
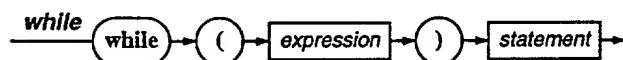
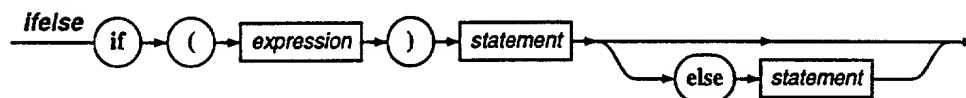
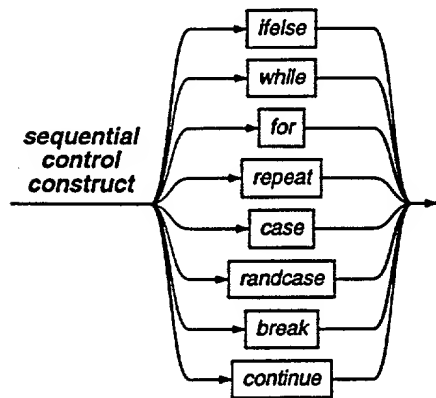
B1.6 Interface Signal Control

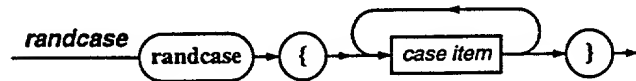
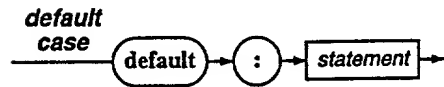
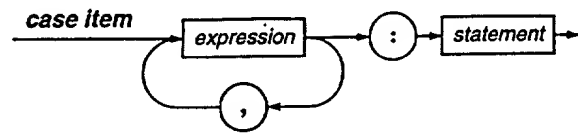




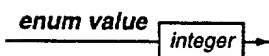
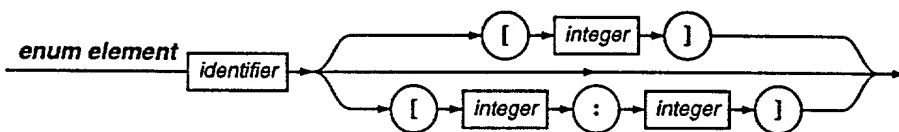
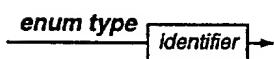
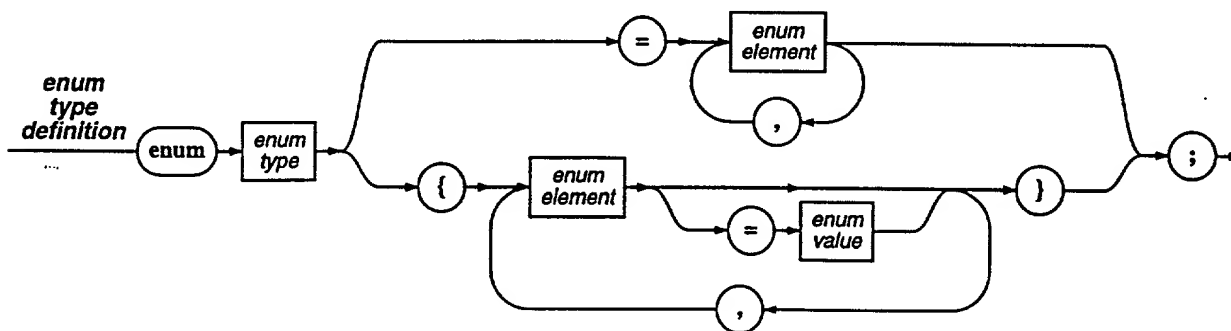


B1.7 Sequential Control



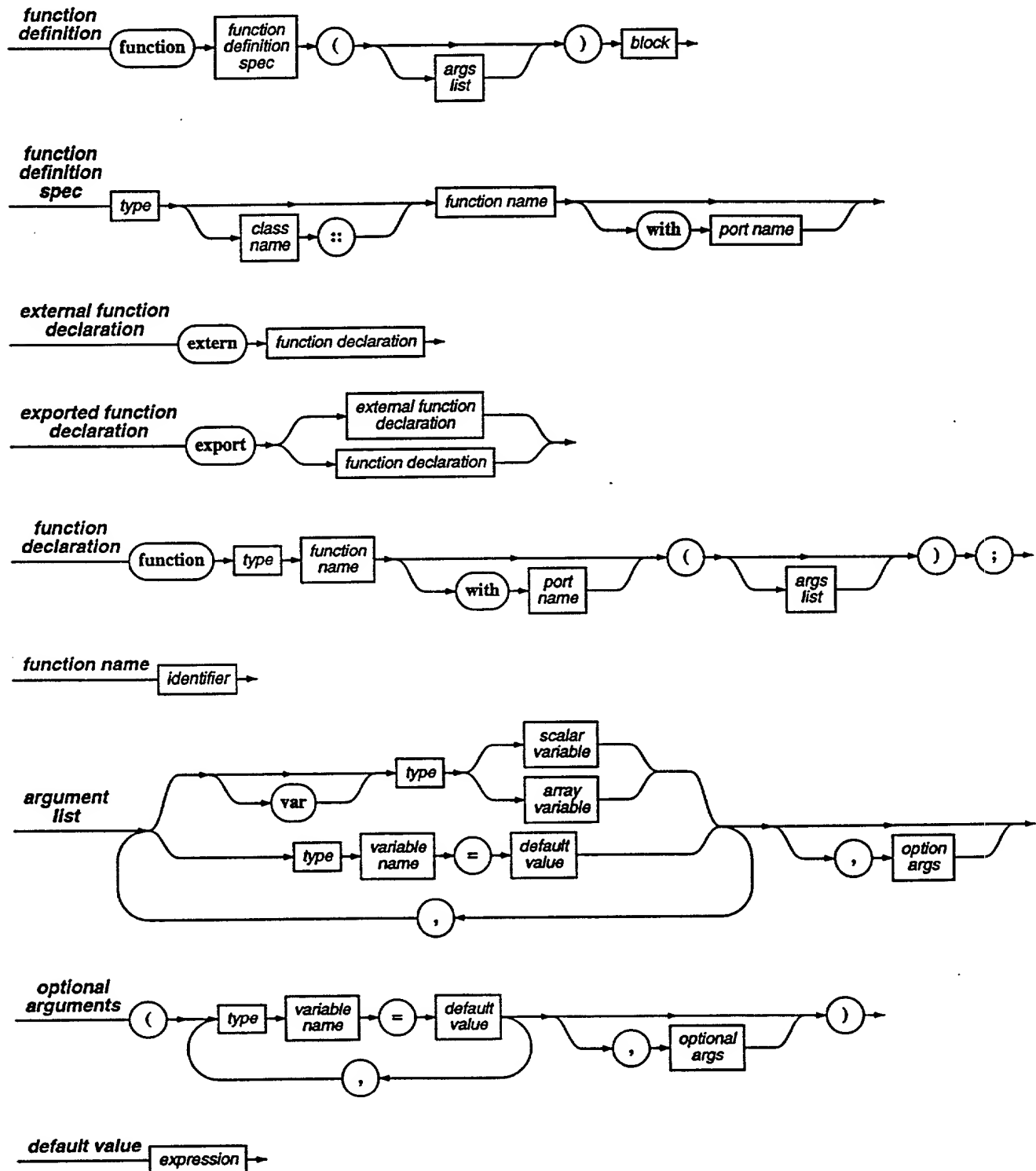


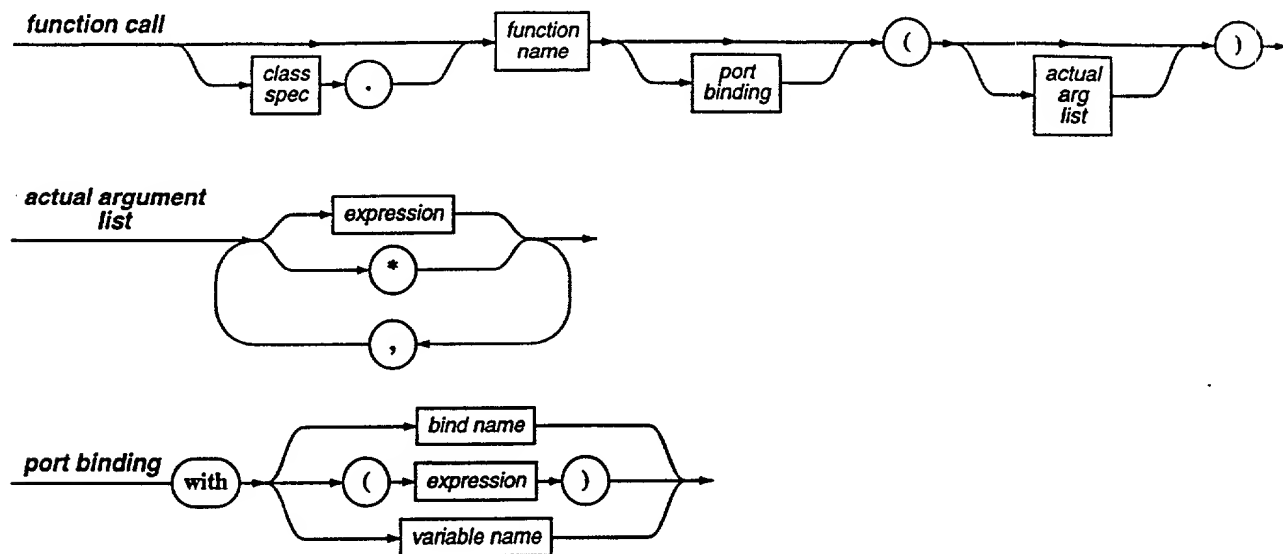
typedef definition → typedef → class → class name →



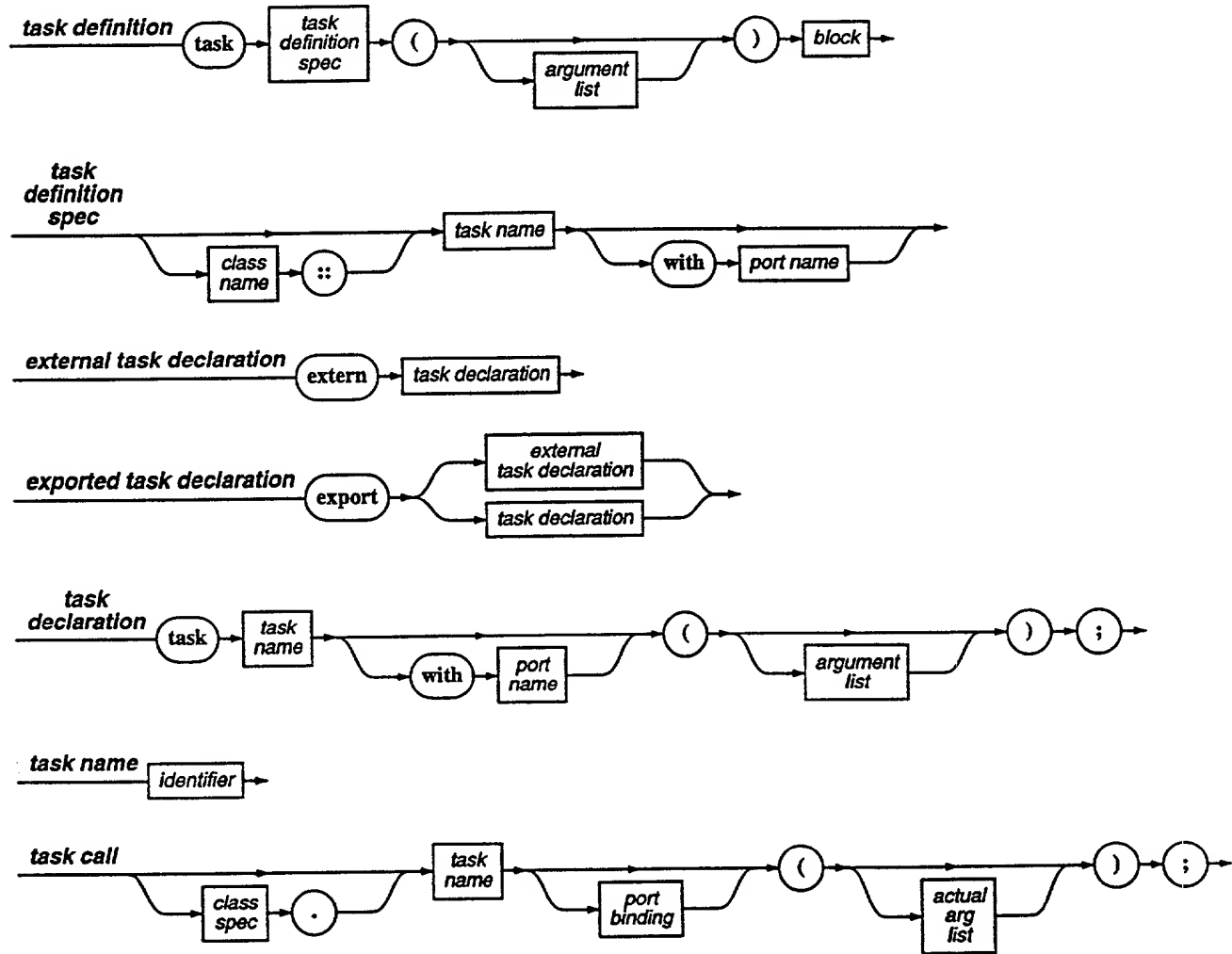
(387)

B1.9 Functions

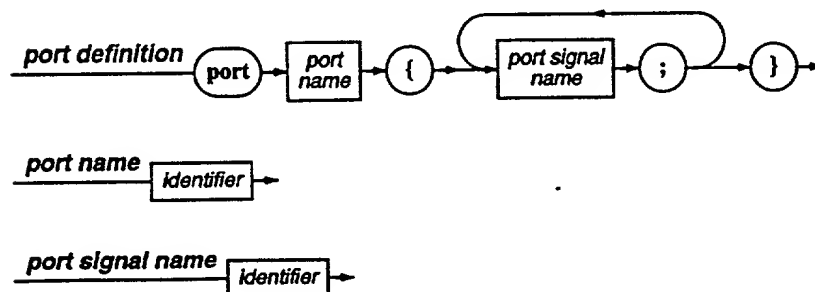




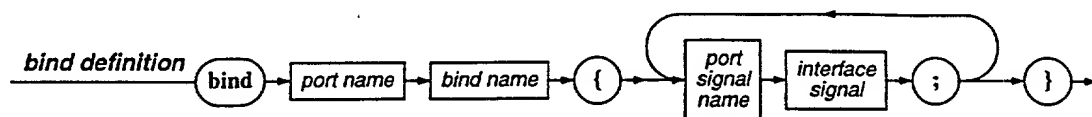
B1.10 Tasks



B1.11 Ports and Binds

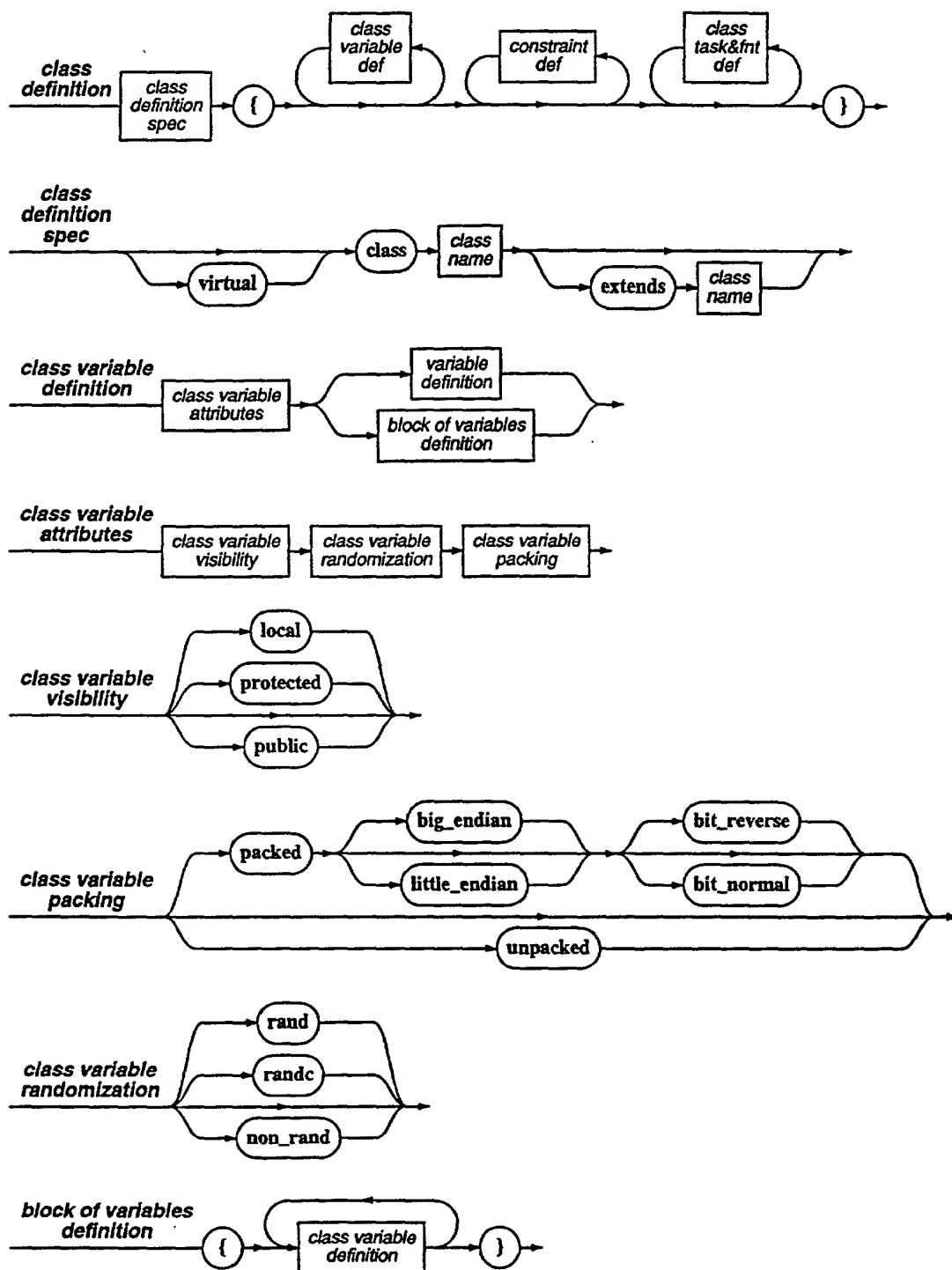


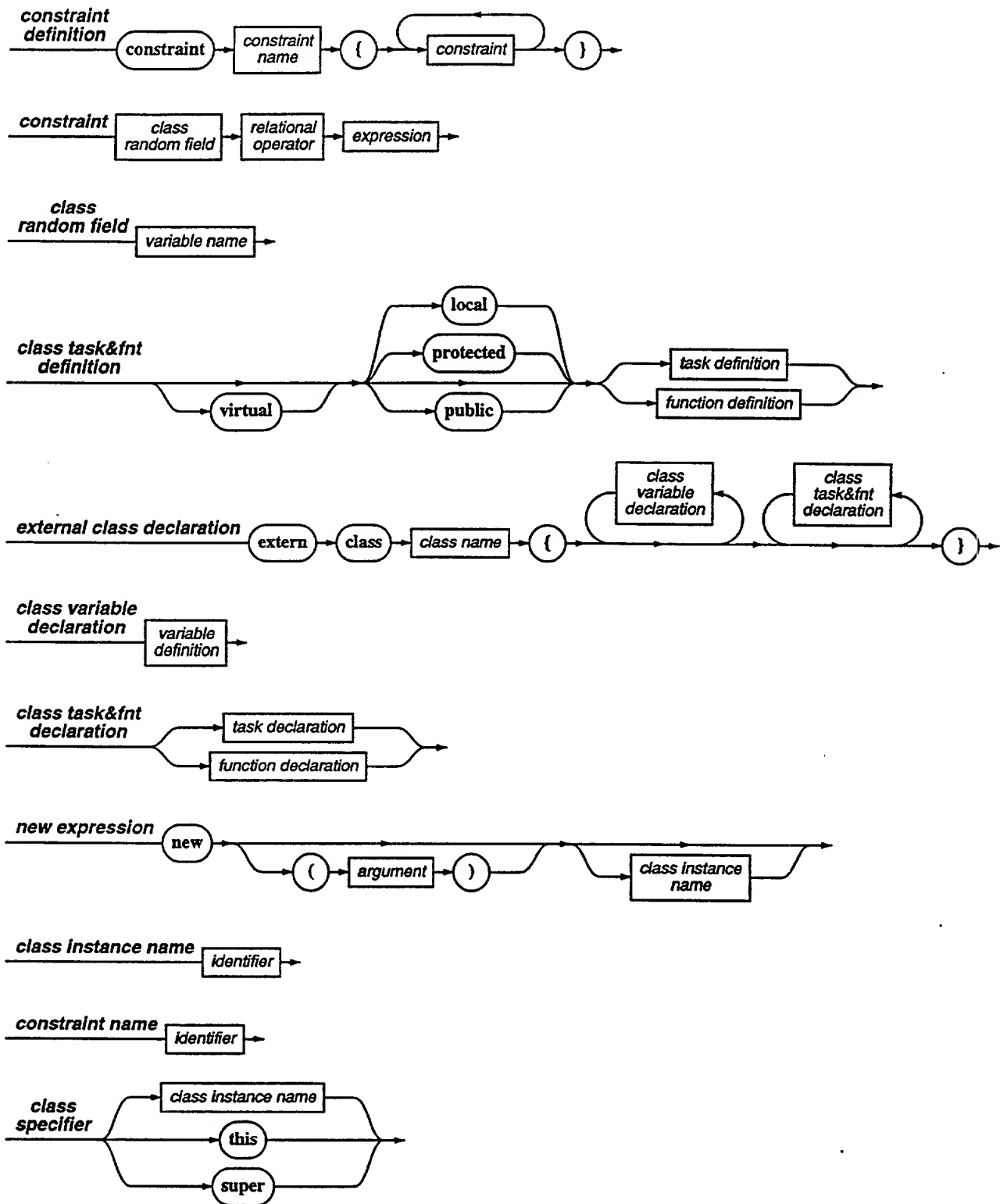
(390)



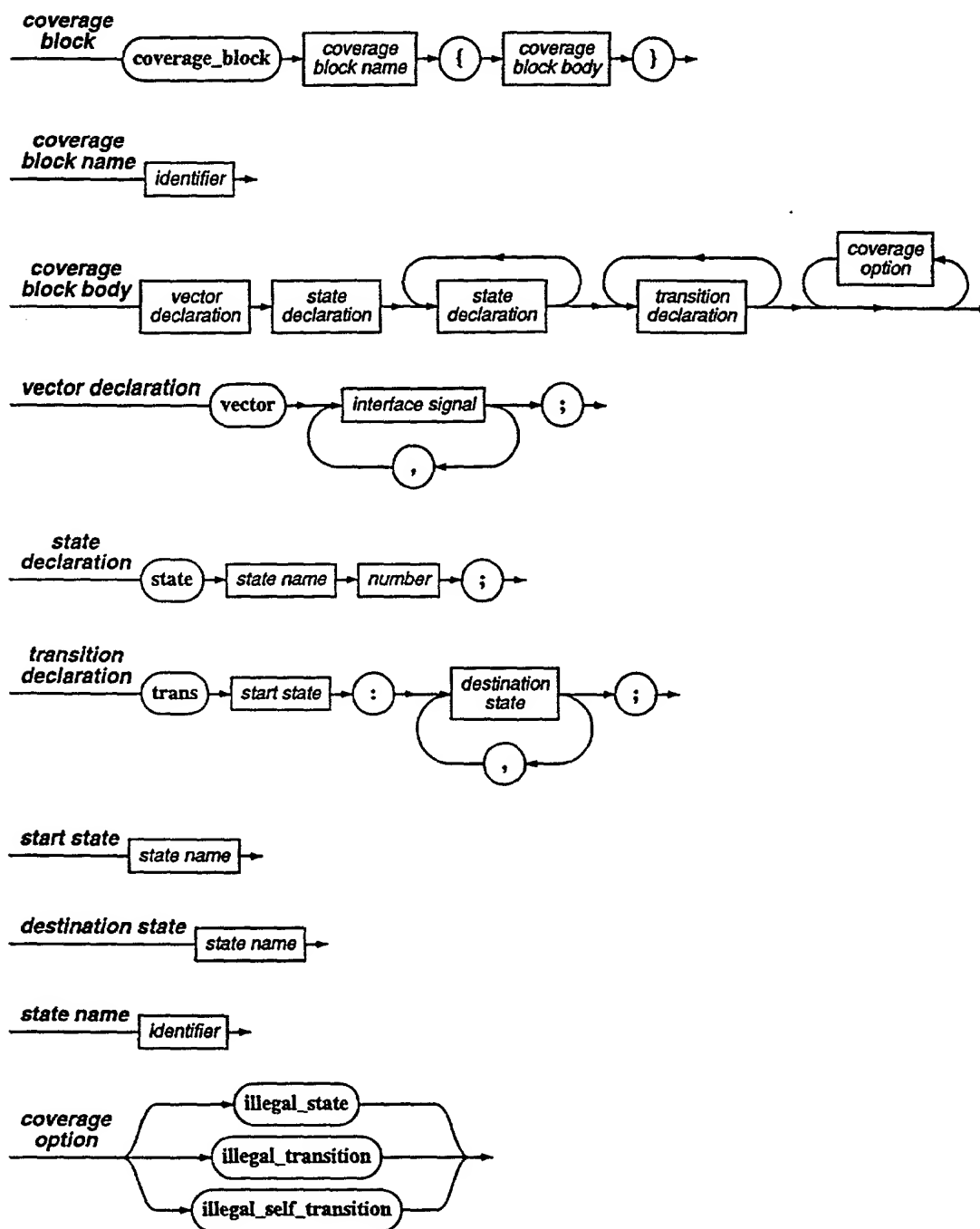
Copyright © 2003 Synopsys, Inc.

B1.12 Classes

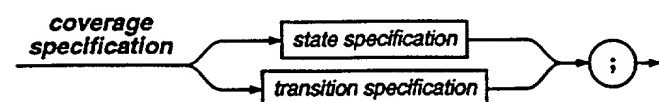
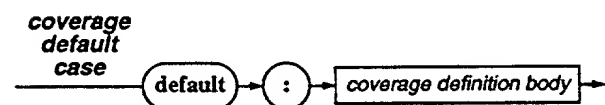
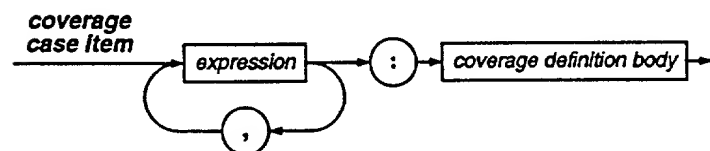
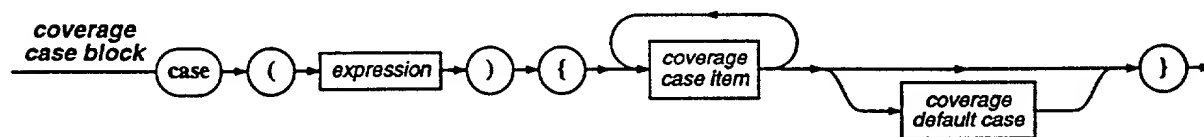
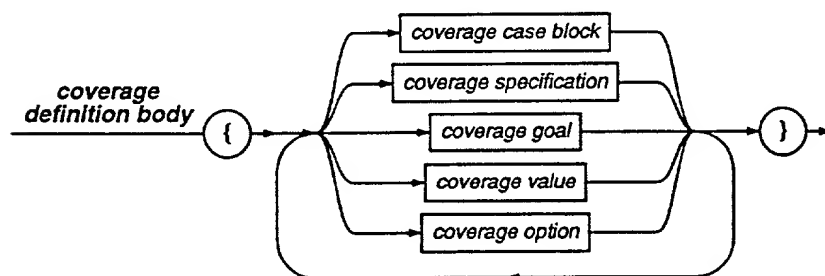
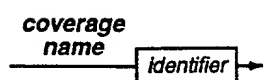
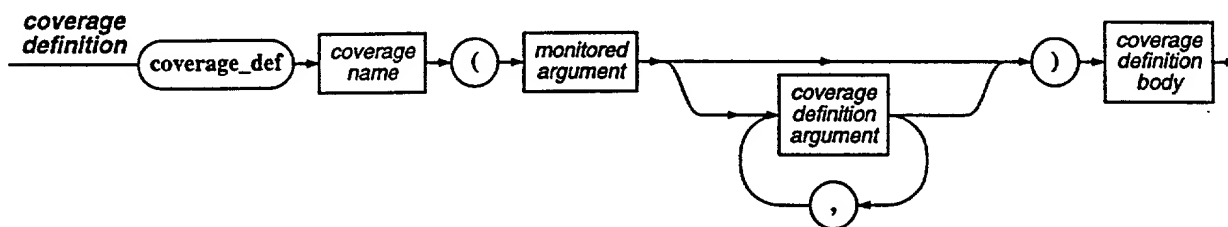


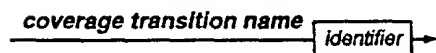
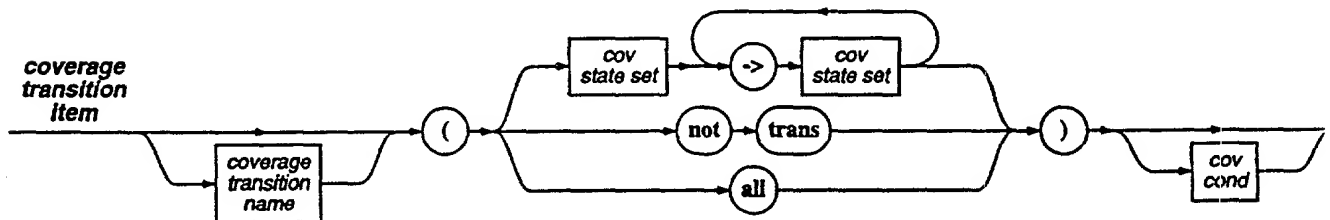
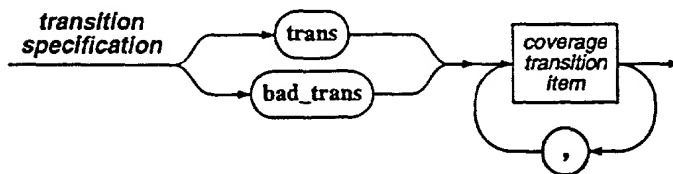
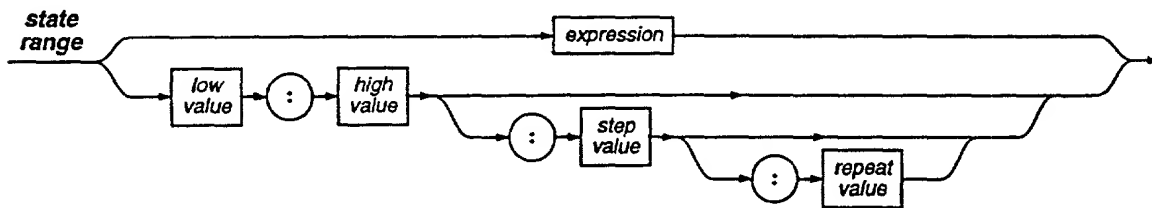
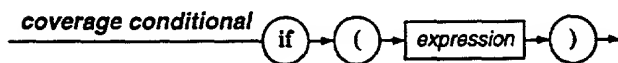
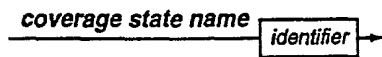
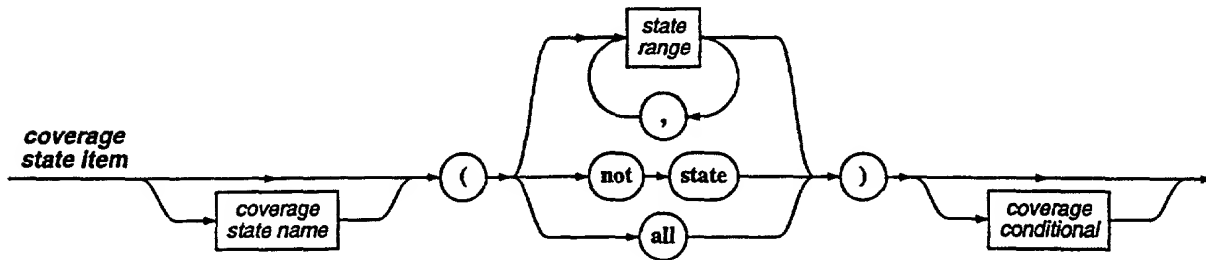
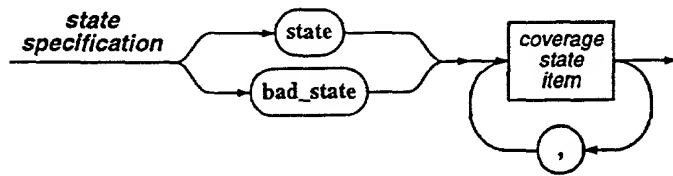


B1.13 Coverage Blocks

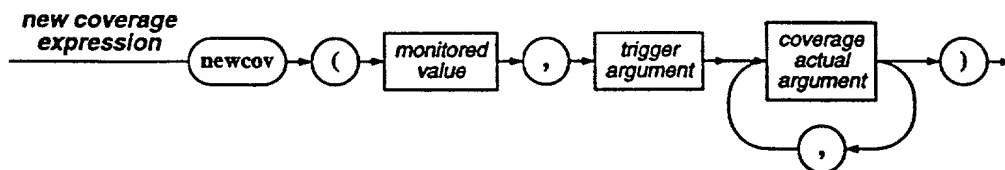
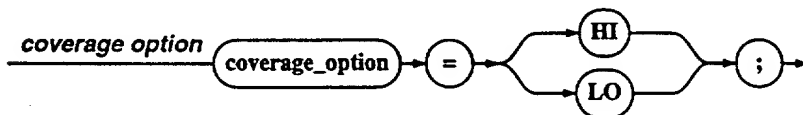
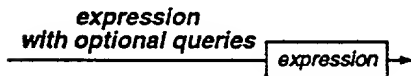
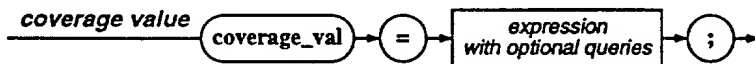
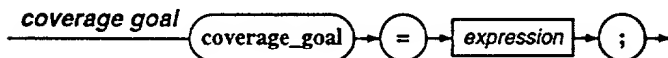
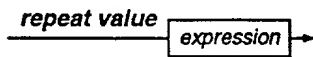
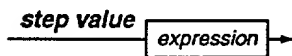
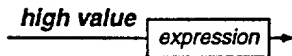
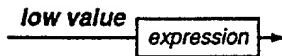
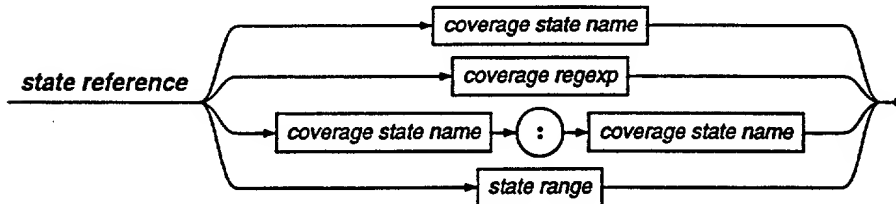
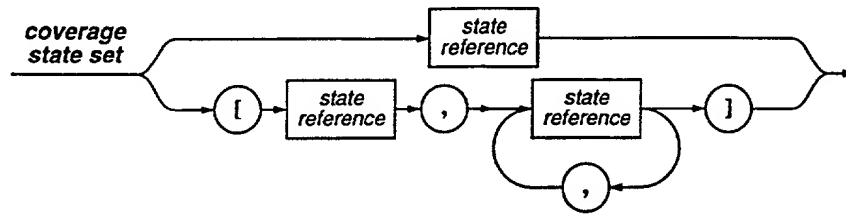


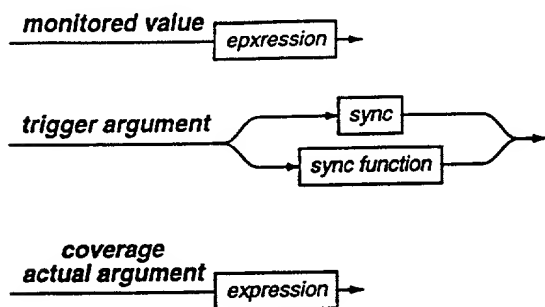
B1.14 Coverage Objects Definition





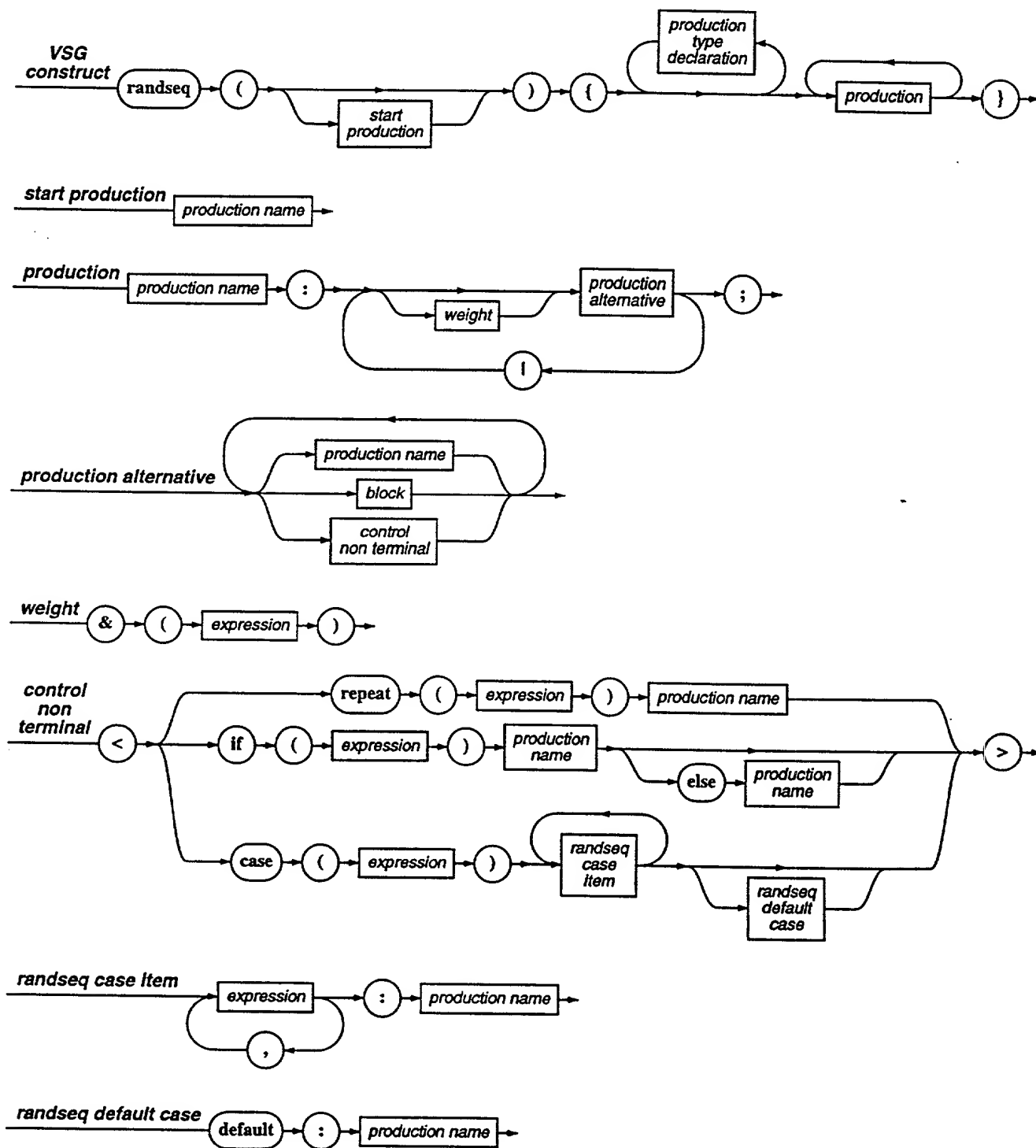
(396)

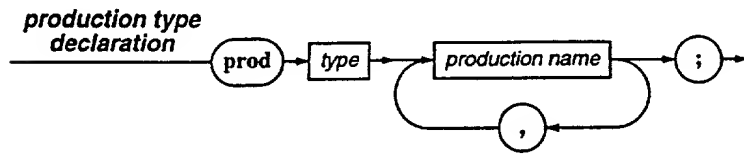




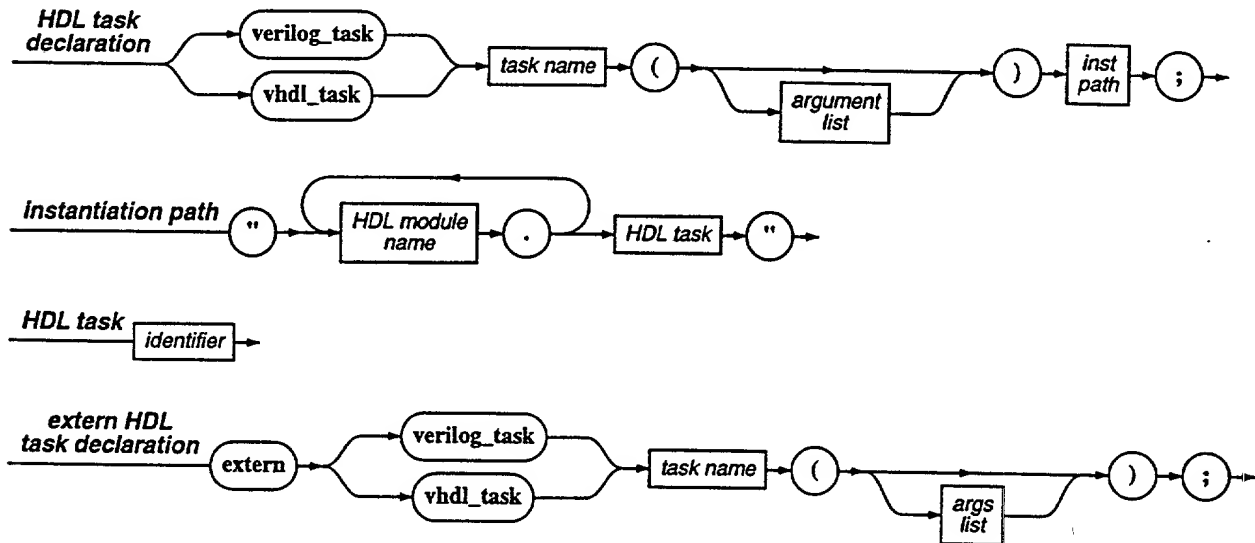
Copyright © 2000 Synopsys Inc.

B1.15 Vera Stream Generator

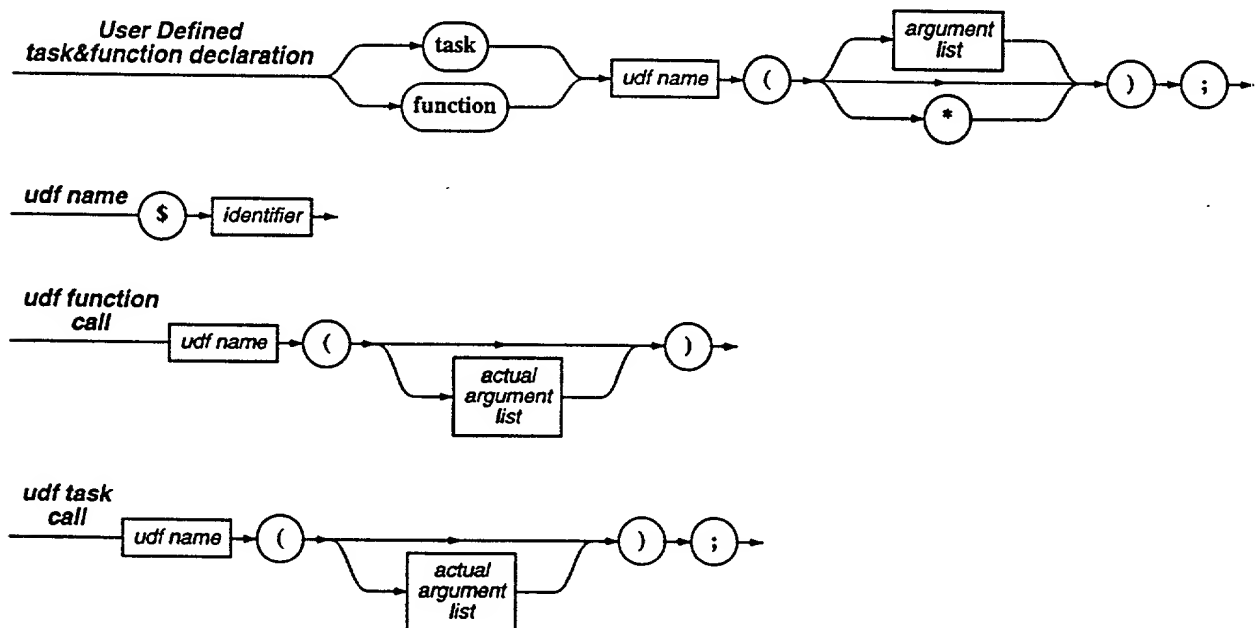




B1.16 HDL Tasks



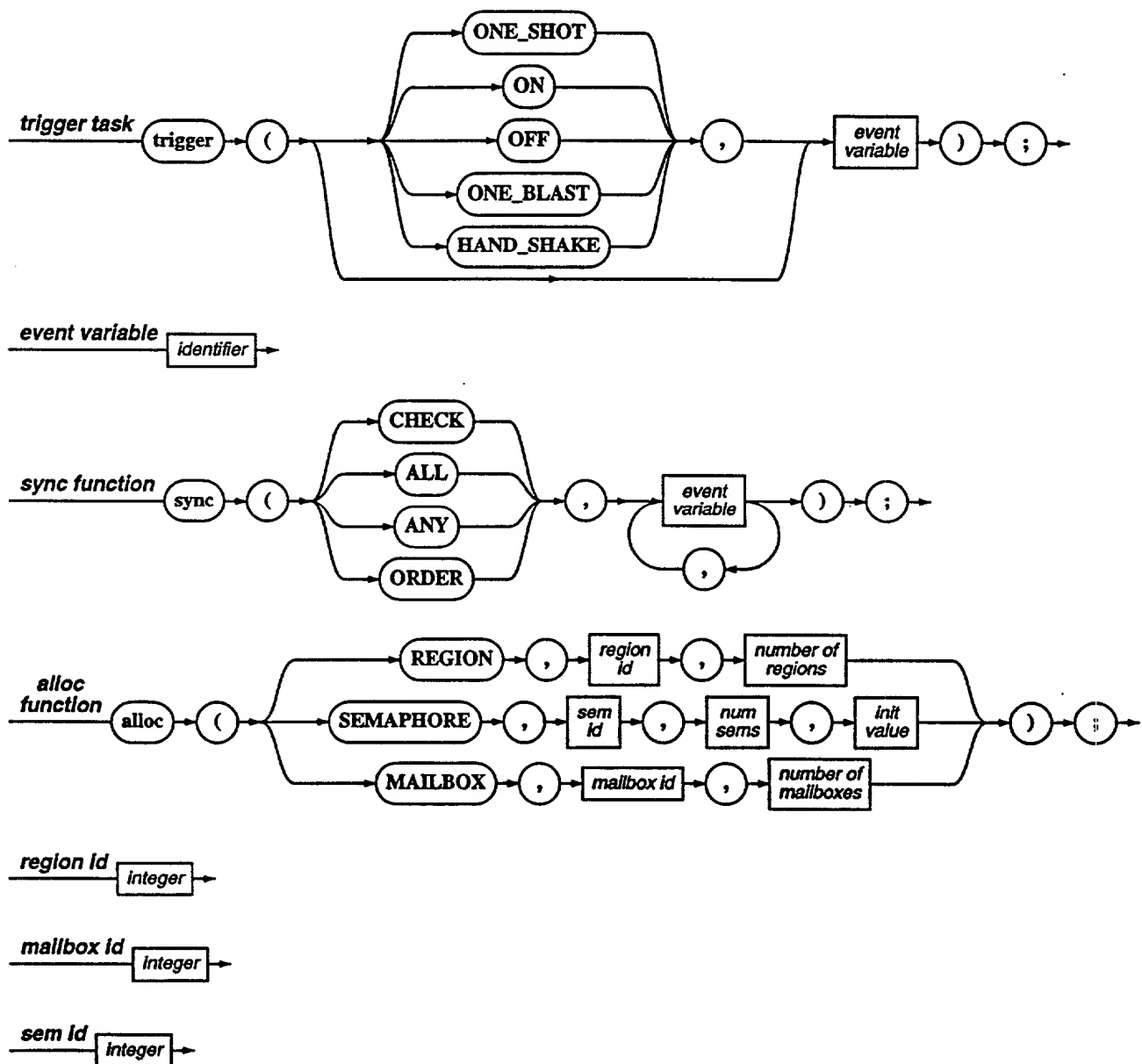
B1.17 UDF Declarations



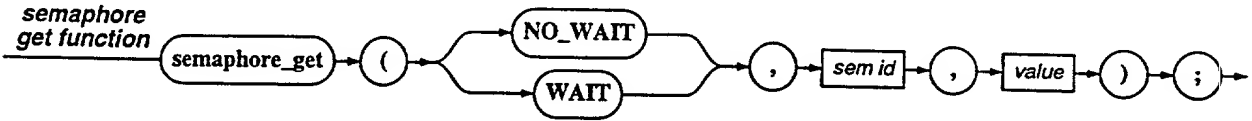
(401)

B2. Vera System Tasks & Functions

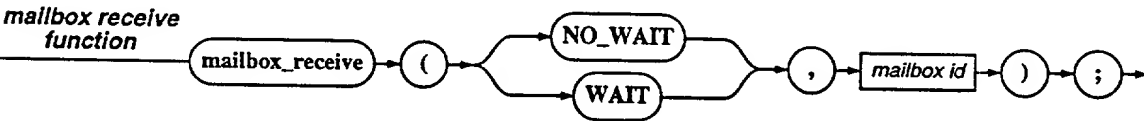
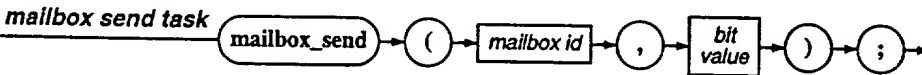
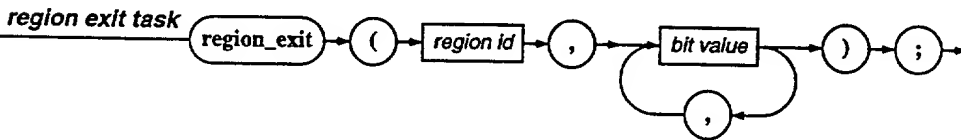
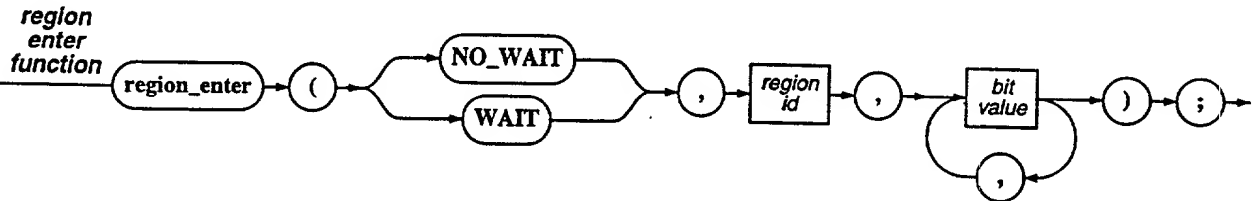
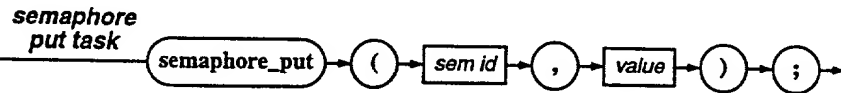
B2.1 Concurrent Control



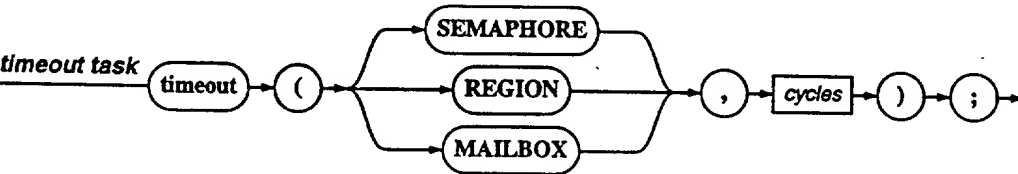
init value **integer** →



value **integer** →



bit value **expression** →

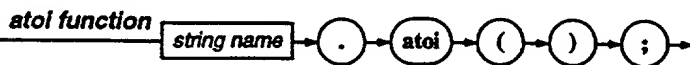
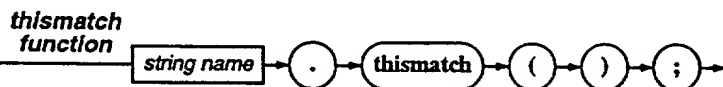
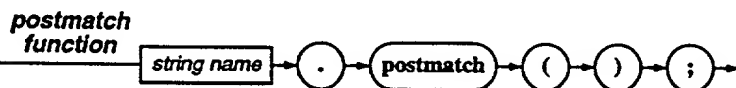
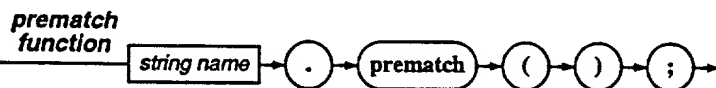
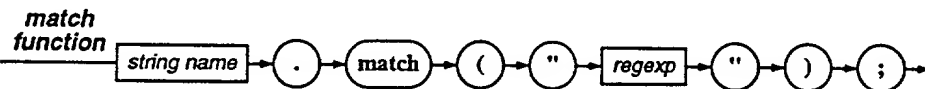
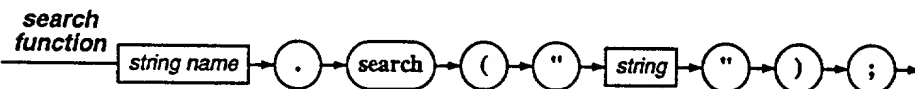
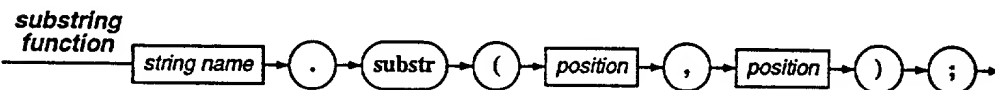
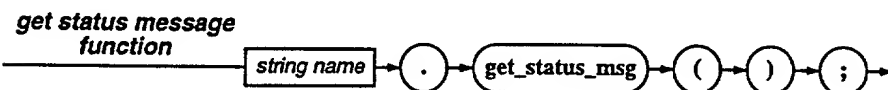
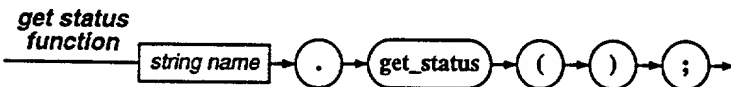
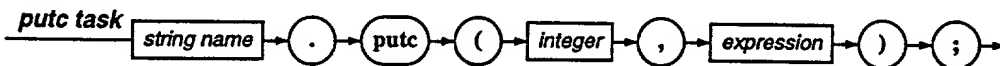
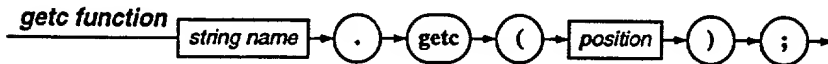


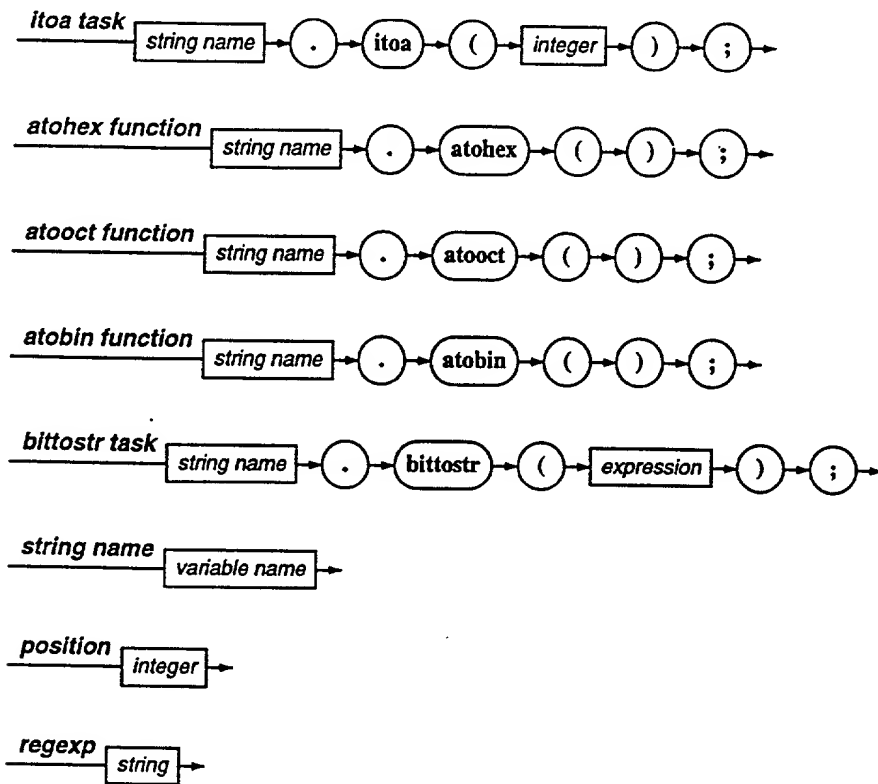
cycles **integer** →



(403)

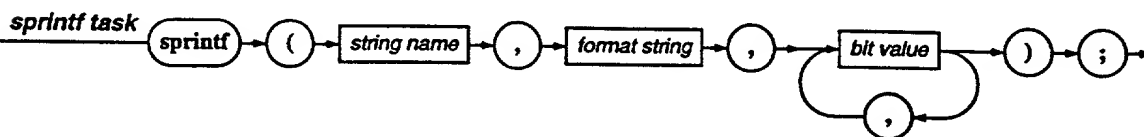
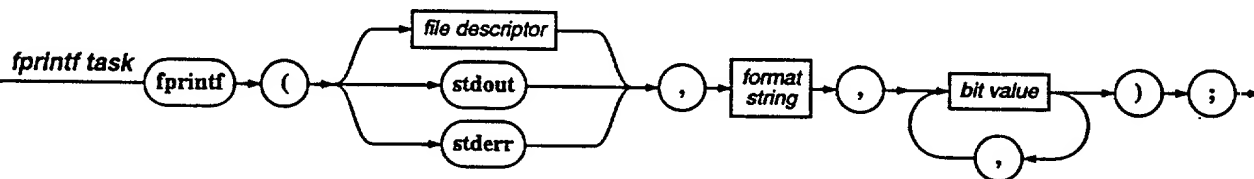
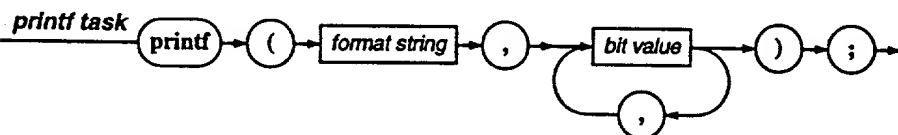
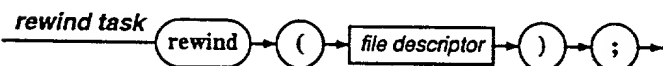
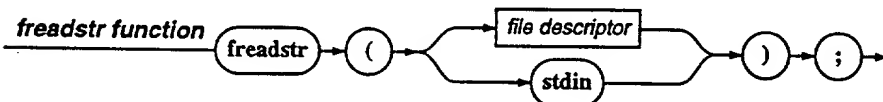
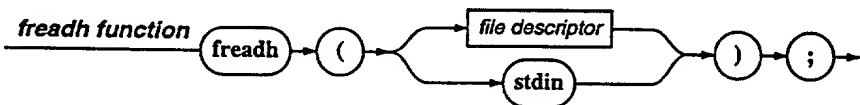
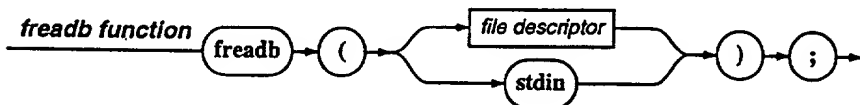
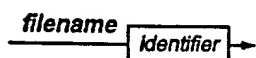
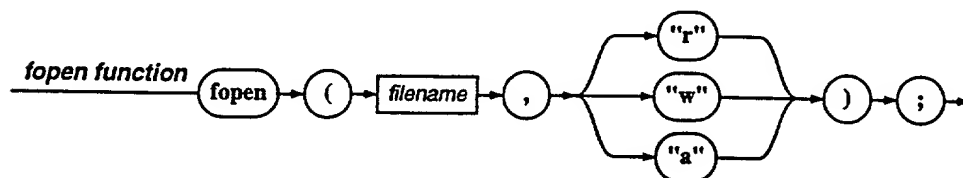
B2.2 String operations



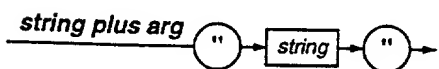
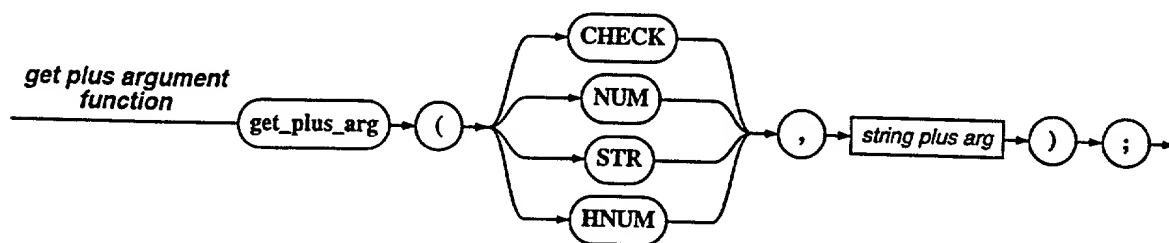
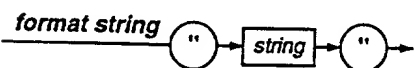
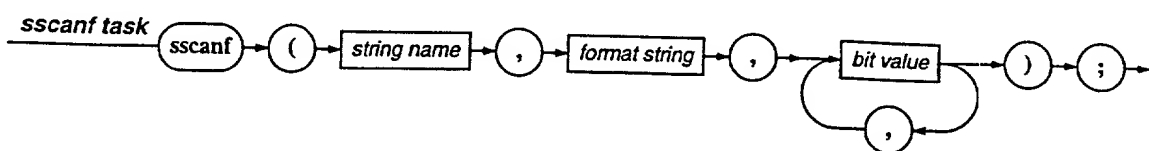


(405)

B2.3 Input/Output

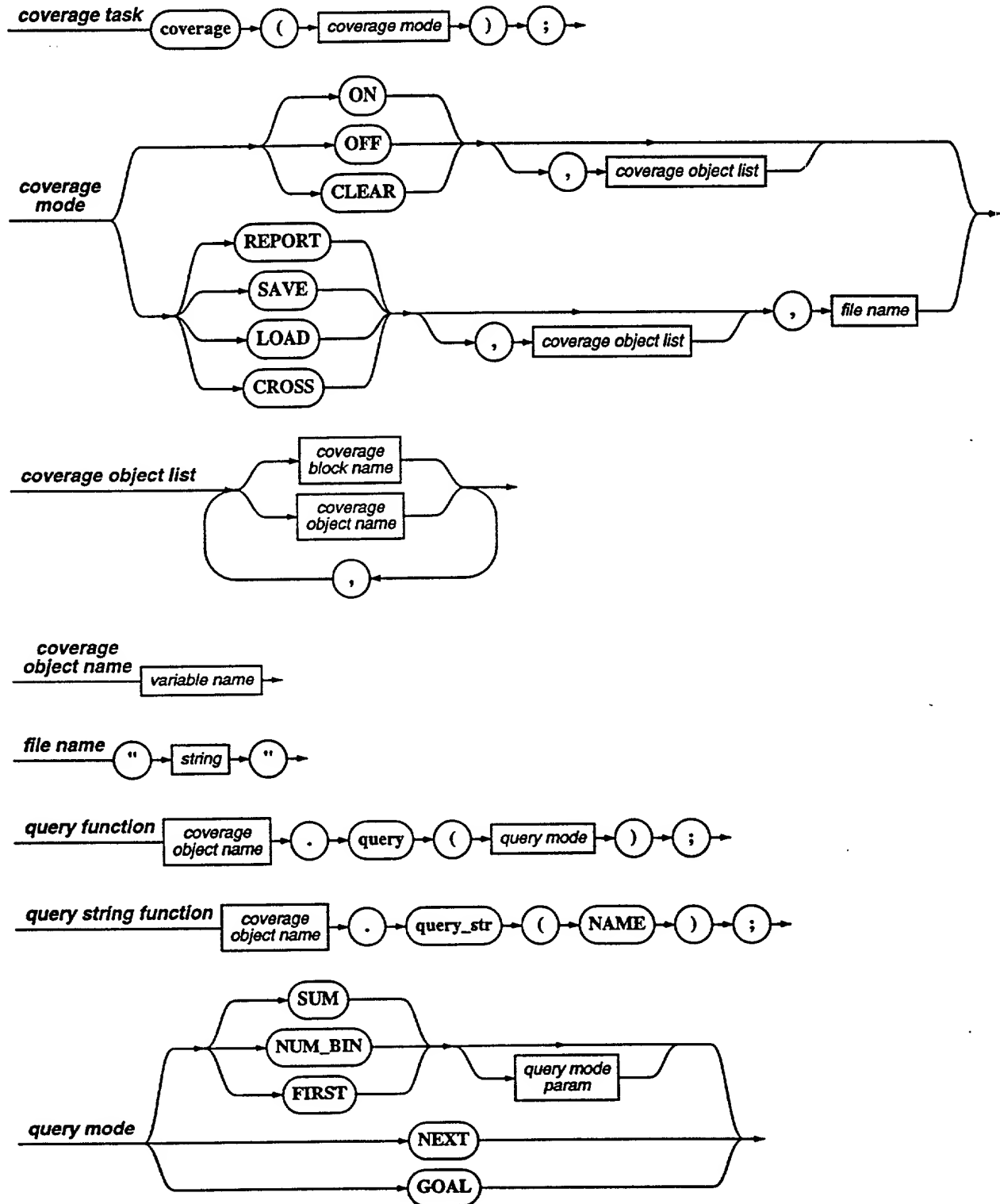


(406)

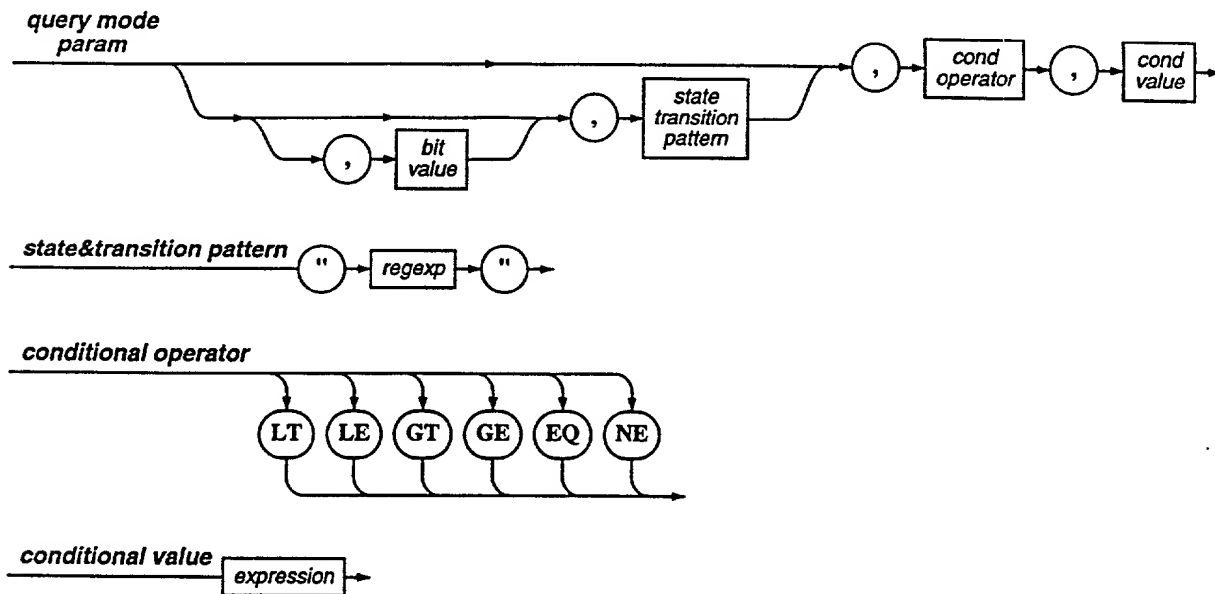


(407)

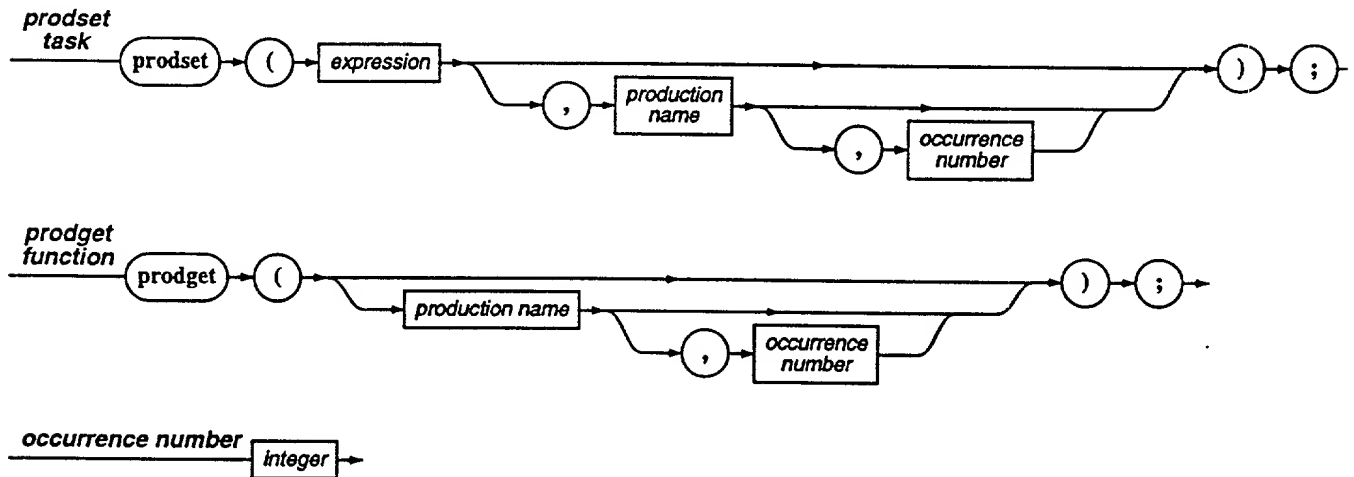
B2.4 Coverage Control



(408)

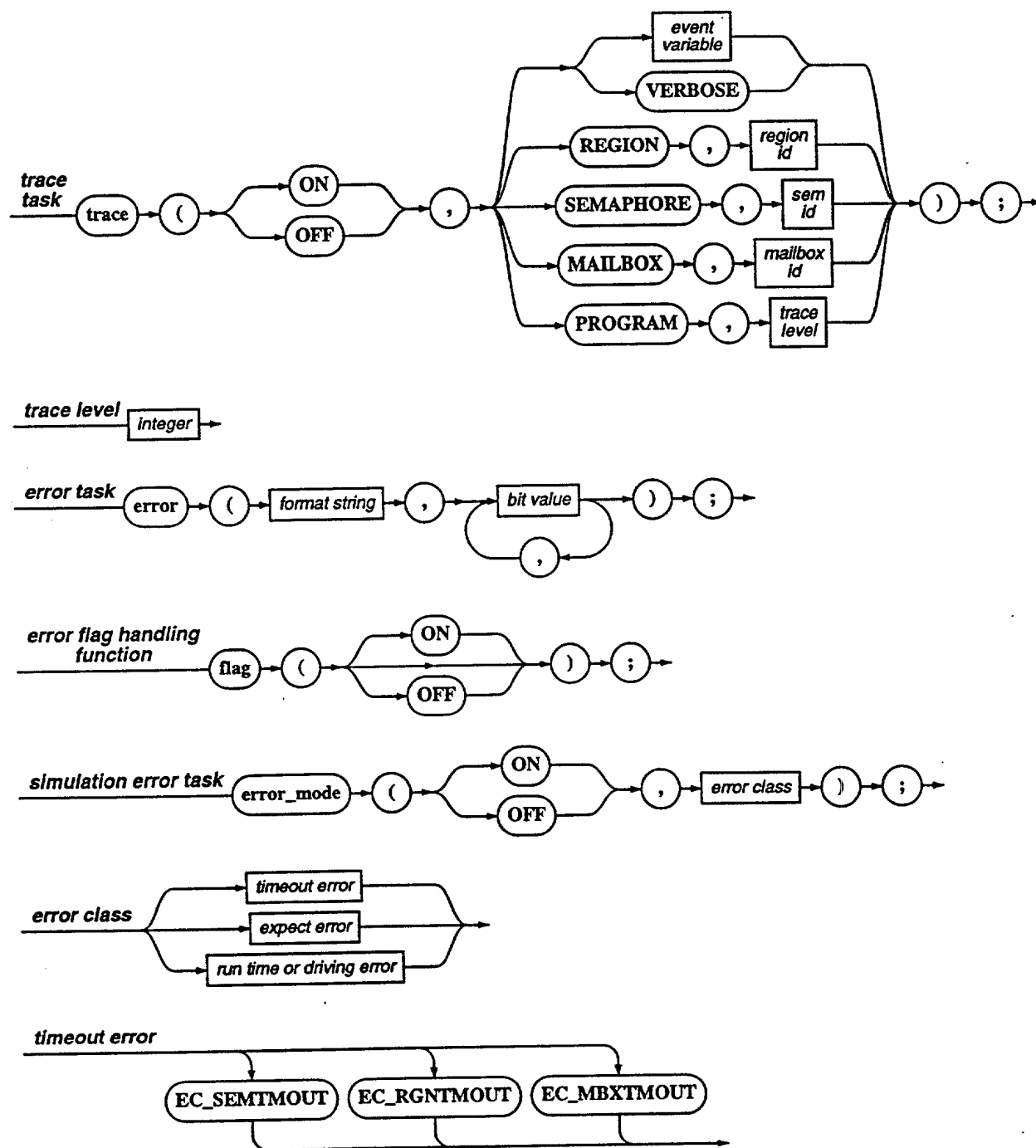


B2.5 VSG Value Passing Support

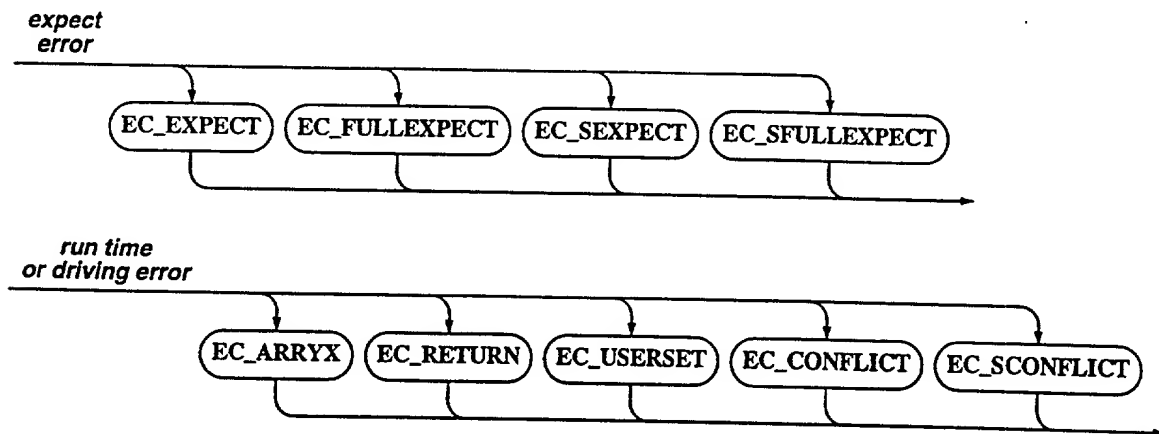


(409)

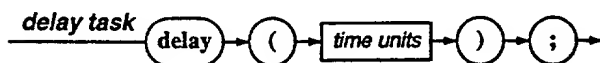
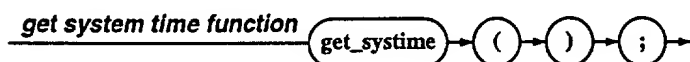
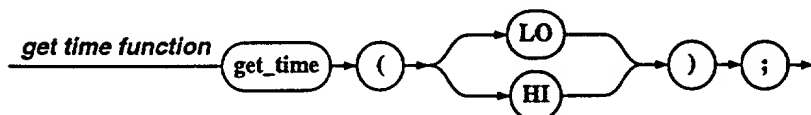
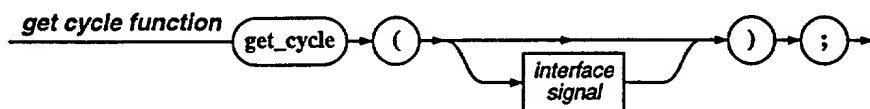
B2.6 Debug Support



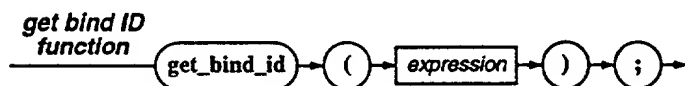
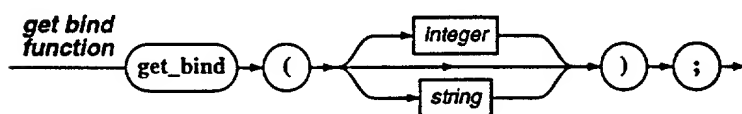
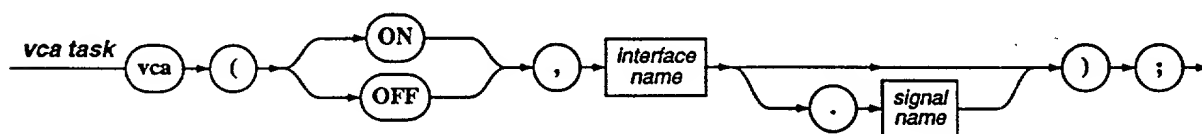
(410)



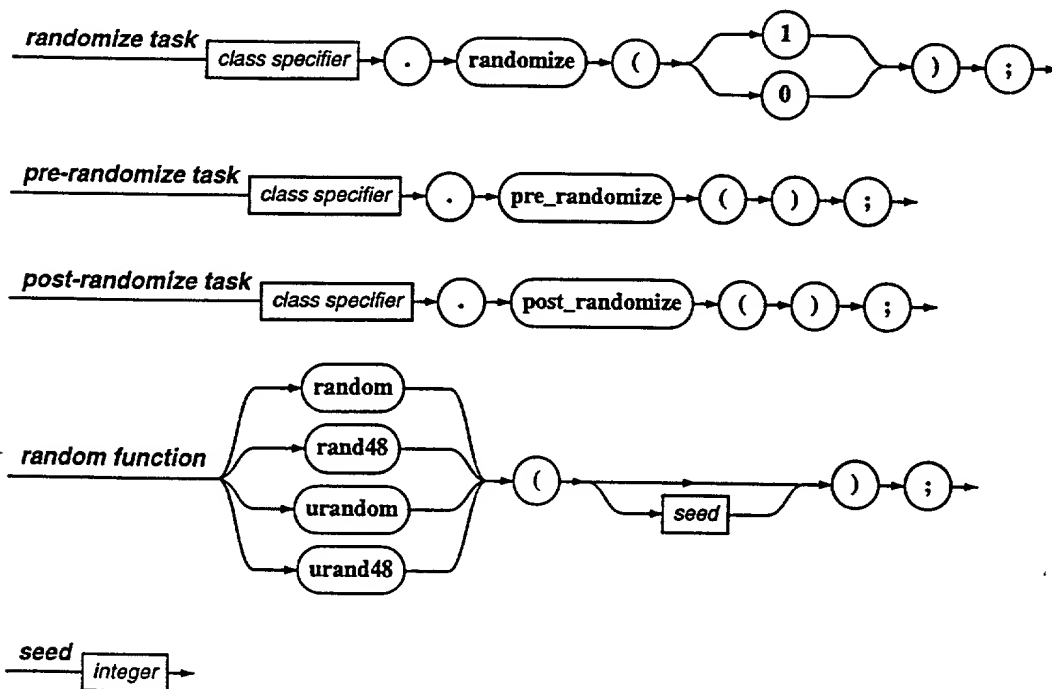
B2.7 Simulation Timing



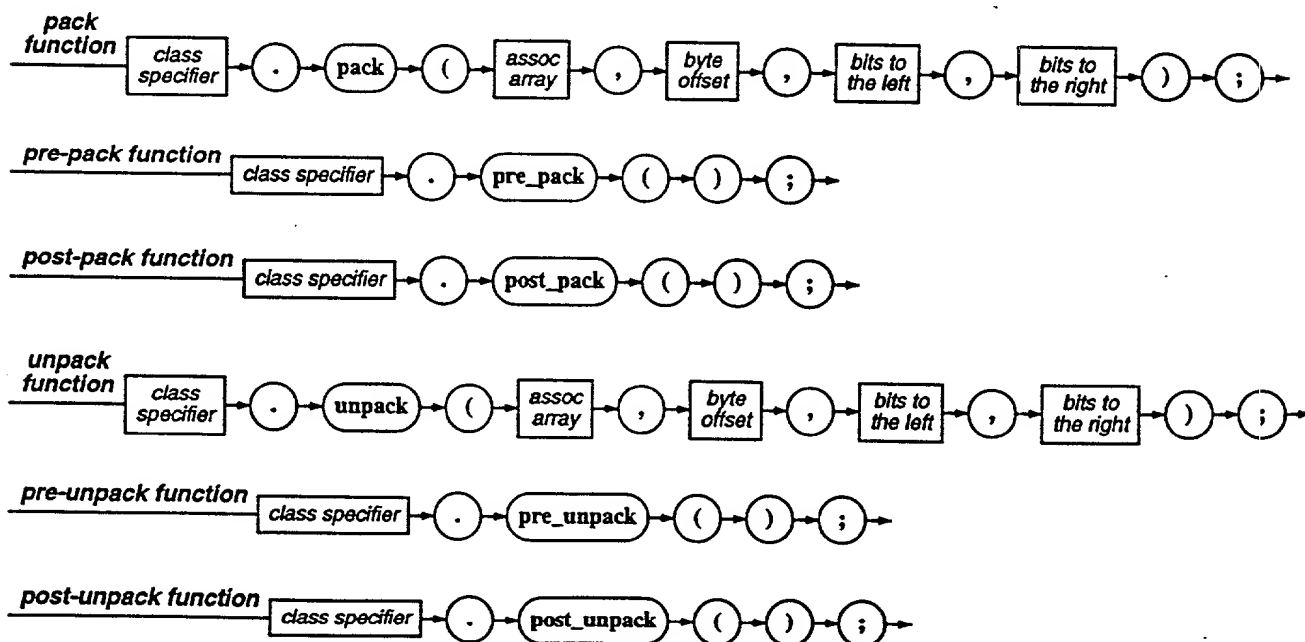
B2.8 Signal Control

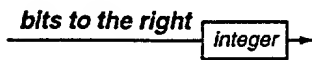
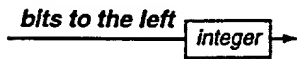
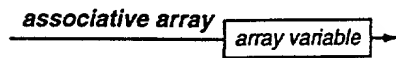


B2.9 Automated Stimulus Generation

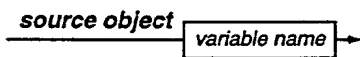
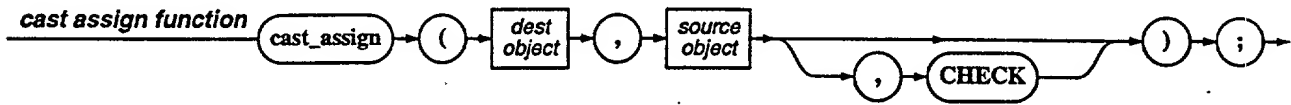


B2.10 Data compaction

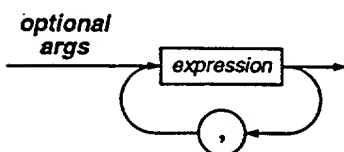
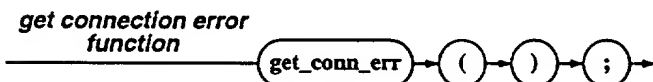
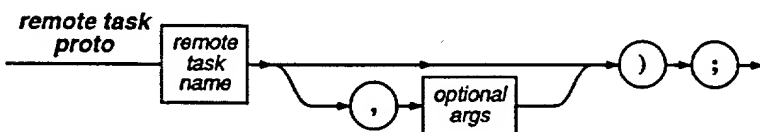
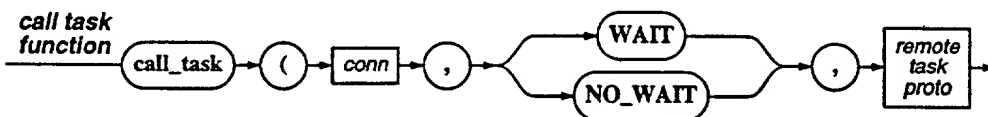
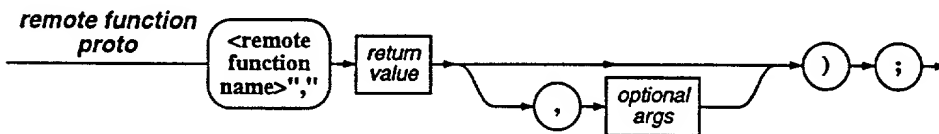
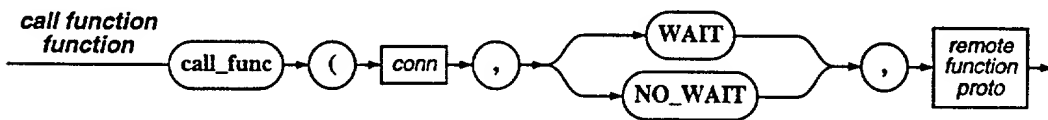
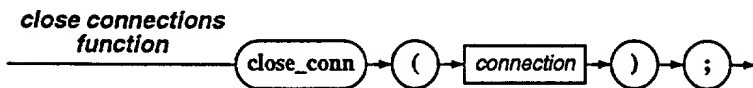
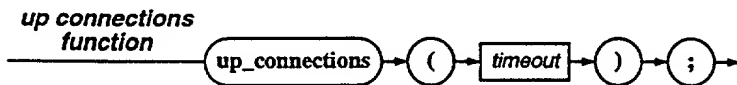
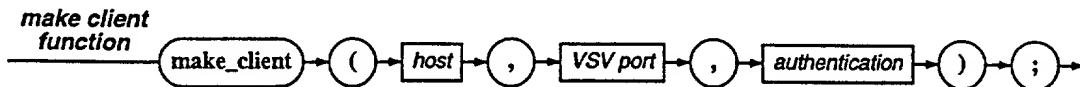
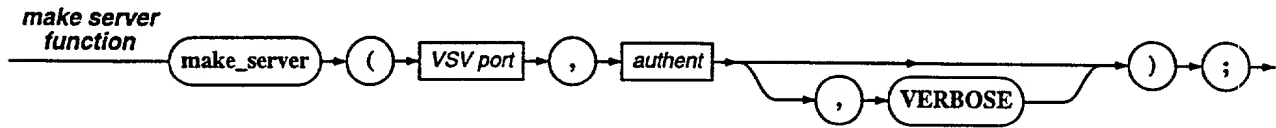


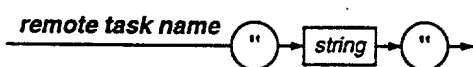
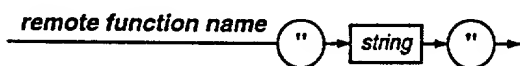
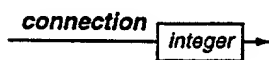


B2.11 Polymorphism support

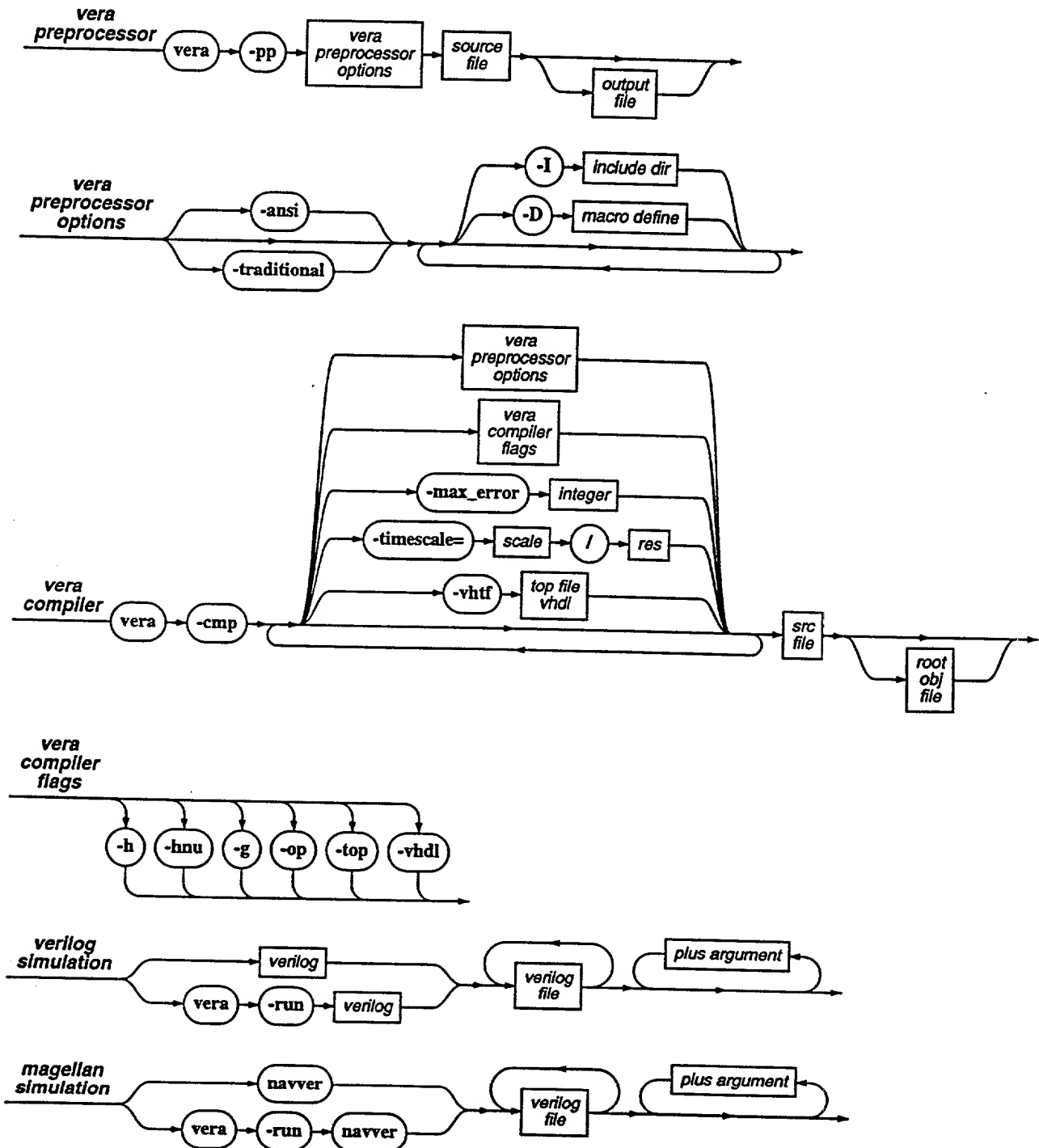


B2.12 VERA-SV API

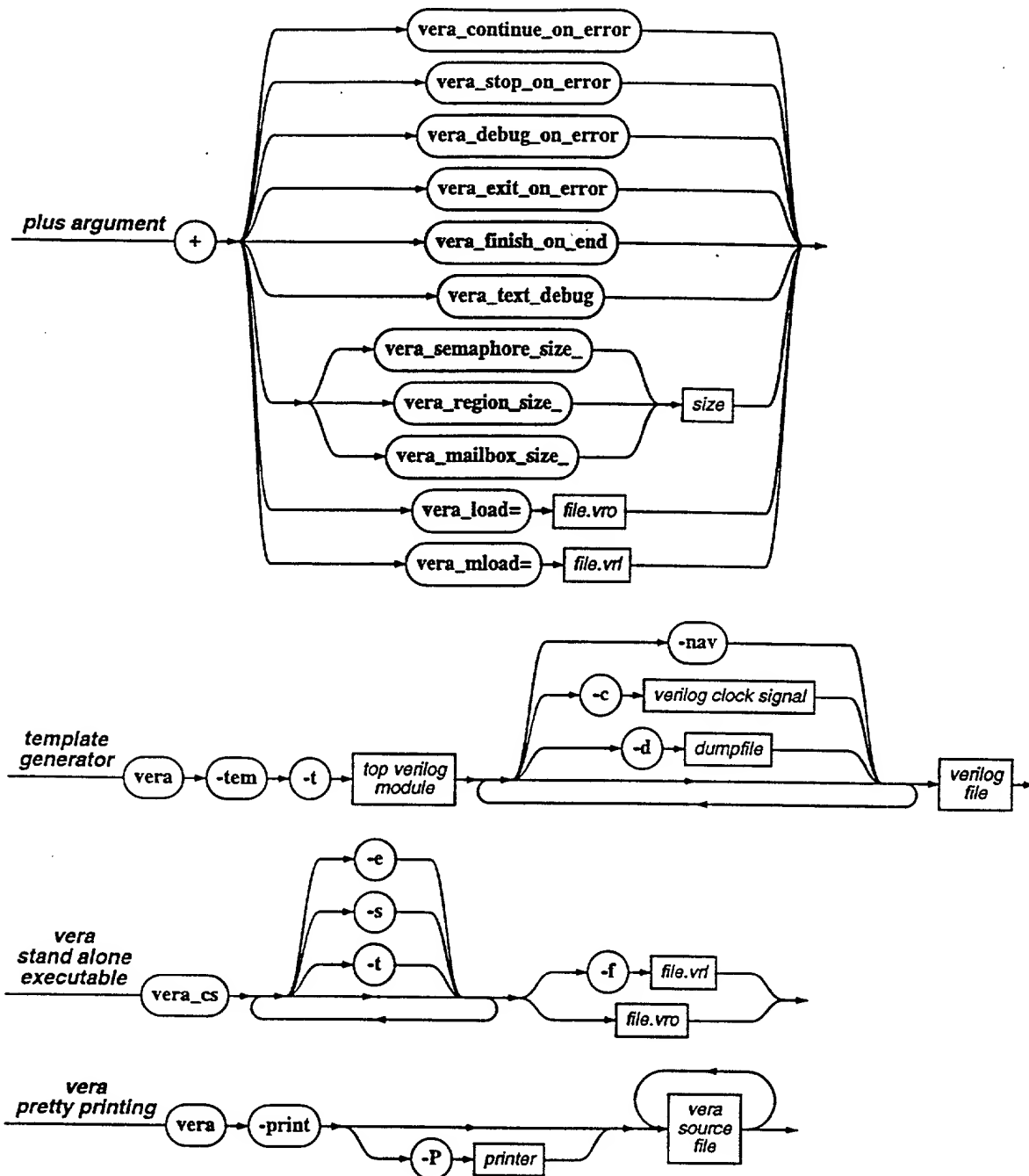




B3. Vera Simulator



(417)



Appendix C. Perl Expressions

Character	Description
.	Matches any character (other than a newline)
(...)	Groups a series of elements
+	Matches the preceeding pattern 1 or more times
?	Matches the pattern 0 or 1 time
*	Matches the pattern 0 or more times
[...]	Matches a character from the enclosed set
[^...]	Matches a character from the negated set
(... )	Matches one of the alternatives
^	Matches the beginning of a string
\$	Matches the end of a string
{n,m}	Matches the pattern from n to m times
{n}	Matches the pattern exactly n times
{n,}	Matches the pattern at least n times
\n \t etc.	Matches new-line, tab, etc.
\b	Matches on a word boundary
\B	Matches inside word boundary
\d	Matches a digit
\D	Matches a non-digit
\s	Matches white-space
\S	Matches non-white-space
\w	Matches an alphanumeric character
\W	Matches a non-alphanumeric character

Copyright 1999 Synopsys Inc. All rights reserved. This document is the property of Synopsys Inc. and is not to be distributed outside the company.

Index

Symbols

!in 187

+vera_continue_on_error 345

+vera_debug_on_error 345

+vera_exit_on_error 345

+vera_finish_on_end 345

+vera_load 344

+vera_mailbox_size 346

+vera_mload 344

+vera_region_size 346

+vera_semaphore size 346

+vera_stop_on_error 344

+vera_text_debug 345

A

alloc() 122, 125, 128

arrays 48–50

 associative arrays 49–50

 setting maximum number of elements 341

 support functions 49

assoc_index() 49

associative arrays

 See also arrays

async 90

asynchronous operations 89–91

 async 90

 sub-cycle delays 90

atobin() 57

atohex() 57

atoi() 56

atooct() 57

B

backref() 55

bad_state 228

bad_trans 231

begin/end 42

bidirectional signals 67

big_endian 196

binary numbers 45

bind 70–79

 direct binding 74

 dynamic binding 75–79

 runtime bind declaration 76

 runtime signal mapping 77–79

 void binds 75

 support functions 71–72

bind_vars 46, 72–73

bit_normal 196

bit_reverse 196

bits 46

bittostr() 58

boundary conditions 194–196

boundary() 195

break 109

 usage in VSG 211

breakpoint 94

C

case 106

 usage in VSG 210

 use in coverage objects 234

cast_assign() 47, 161

classes 154–168

 assignment 158

 class methods 156

 class properties 157

 constructors 156

 copying 158

 creating instances 155

 data encapsulation 162

 external classes 167

 inheritance 159

 local members 162

 polymorphism 166

 protected members 162

 renaming 158

 subclasses 159

 superclass 161

 typedef 168

 virtual classes 164

CLOCK 67

clock 256

clock_alias 256

clocking domains 69

comments 41

compiler

 compile options 340–343

 dynamic loading of object modules

348–349
 general options 338–340
 modular compilation 346–348
 running Vera stand-alone 349–351
 running Vera with HDLs 343–346
 compiler switches
 -alim 341
 -ansi 341
 -c 309
 -cmp 339
 -D 341
 -d 310
 -e 350
 -f 350
 -g 341, 353
 -h 342
 -hnu 342
 -I 339
 -i 342
 -ip 342
 -max_error 342
 -nav 310
 -pp 339
 -proj 339
 -q 343
 -run 339
 -s 350
 -t 310, 350
 -tem 340
 -timescale 343
 -top 343
 -v 340
 -vcon 340
 -vhdl 343
 concatenation 63
 conditional compilation 337–338
 configuration files 254, 255–257
 clock 256
 clock_alias 256
 connect 256
 timescale 255
 veritask 257
 connect 256
 constraint blocks 186–190
 active and inactive constraints 188
 boundary conditions 194–196
 constraint errors 188
 distribution sets 187–188

static constraint blocks 190
 void constraints 189
 constraint_mode() 188
 contexts 354
 continue 110
 usage in VSG 212
 coverage
 backward compatibility 244–245
 coverage administration 238–240
 coverage declarations 225–232
 coverage goal 233
 coverage objects 224–235
 expressions 225
 initialization 234
 coverage options 233
 CROSS TRANS 233
 CROSS_TRANS 241
 HI 233, 241
 LO 233
 NO_OVERLAP 233
 coverage reports 239
 coverage_def 224
 cross coverage 240–244
 illegal state declarations 228
 illegal transition declarations 231
 instantiation 235
 overview 223
 state declaration 225–228
 transition declarations 228–232
 state transition 229
 triggering coverage objects 235
 coverage_def 224
 coverage_goal 233
 coverage_val 232
 CROSS_TRANS 233, 241

D

debugger
 accessing context information 358–359
 accessing source files 359–360
 debugging environment control 362
 displaying and updating data 357–358
 execution control 356–357
 expansion windows 364
 expressions 354
 graphical debugger 363–364
 miscellaneous commands 362

- radix 362
- text-based commands 354–363
- watch commands 360–361
- watch windows 364
- window 363
- debugger commands
 - , 356
 - „ 356
 - . 356
 - .. 356
 - assign 357
 - break 356
 - clear break 356
 - close 362
 - context 359
 - down 359
 - echo 359
 - end 359
 - help 362
 - limit array 357
 - list 360
 - print 357
 - quit 362
 - restore 362
 - save 362
 - show break 357
 - show context 358
 - show files 360
 - show vars 358
 - show watch 361
 - source file 359
 - up 359
 - watch add 360
 - watch change 361
 - watch insert 361
 - watch remove 361
 - where 359
- decimals 45
- delay() 82, 90, 350
- dist 188
- distribution sets 187–188
- drive 81–83
 - blocking and non-blocking drives 82
 - implicit synchronization 89
 - soft and strong drives 83
 - void drives 83
- E
 - enumerated types 47, 59–61
 - increment and decrement operations 60
 - mapping Vera types to VHDL 324
 - use in numerical expressions 60
 - error() 146
 - error_mode() 147
 - event variables 119–120
 - bidirectional variables 120
 - events 47, 116–122
 - disabling events 121
 - event variables 119–120
 - merging events 121
 - sync() 118–119
 - timeout() 130
 - triggers 117–118
 - exit() 149
 - expansion windows 364
 - expect 84–87
 - full expect 85–86
 - implicit synchronization 89
 - restricted expect 86
 - simple expect 85
 - strong and soft expects 86
 - void expects 87
 - export 277
 - exporting external tasks 278
 - external declarations 97
- F
 - fclose() 142
 - file inclusion 336
 - file input/output 142–145
 - flag() 146
 - FlexLM 31
 - backwards compatibility 34
 - license file 33
 - LM_LICENSE_FILE 33
 - starting Flex License Manager 34
 - fopen() 142
 - for 108
 - fork and join 111–116
 - shadow variables 116
 - suspend_thread() 115
 - terminate 114
 - wait_child() 113
 - wait_var() 114

fprintf() 142
 freadb() 143
 freadh() 143
 freadstr() 143
 functions 91–92, 93–97
 default arguments 95
 discarding return values 92
 external default arguments 97
 external functions 97
 optional arguments 96

G

get_bind() 71
 get_bind_id() 72
 get_cycle() 150, 225
 get_plus_arg() 151–152
 get_status() 53
 get_status_msg() 53
 get_systime() 150
 get_time() 150, 225, 350
 getc() 52

H

HDL nodes 68–69
 hexadecimals 45
 HI 241

I

identifiers and keywords 42–44
 ifdef 337
 if-else 105–106
 usage in VSG 210
 ifndef 337
 in 187
 inheritance 159
 inout signals 67
 input signals 67
 installation 31
 integers 45
 interface specification 67–68
 clocking domains 69
 HDL nodes 68–69
 signal direction 67
 signal type 67
 signal width 67
 skew 67

itoa() 56

J

join
 See fork.

K

keywords
 See also identifiers and keywords

L

len() 52
 lexical conventions
 comments 41
 statement blocks 42
 white space 41
 lexical convetions
 numbers 44
 licensing 31–34
 FlexLM 31
 lmhostid 32–33
 Vera license queueing 34
 LIT_INTEGER 44
 little_endian 196
 LM_LICENSE_FILE 33
 lmhostid 32–33
 local class members 162

M

m_bad_state 228
 m_bad_trans 231
 m_state 227
 m_trans 230
 mailbox_get() 128
 mailbox_put() 128
 mailbox_receive() 131
 mailbox_send() 131
 mailboxes 127–130
 alloc() 128
 backward compatibility 131
 mailbox_get() 128
 mailbox_put() 128
 mailbox_receive() 131
 mailbox_send() 131
 setting maximum number 323, 346
 timeout() 130

- match() 54
- ModelTech
 - recompiling 37–38
- multiple clocking domains
 - in Verilog 319
 - in VHDL 332
- multiple modules
 - configuration files 254, 255–257
 - generating a multiple module testbench 257–258
 - overview 253–254
 - project files 254, 255
- N**
- negedge 81
- newcov 235
- NHOLD 67
- NO_OVERLAP 233
- not state 228
- not trans 231
- NR 67
- NSAMPLE 67
- null events 47
- numbers 44
- O**
- objects 154–157
 - assignment 158
 - constructors 156
 - copying 158
 - creating instances 155
 - object methods 156
 - object properties 156
 - renaming 158
- octal numbers 45
- operators 61
- output signals 67
- P**
- pack() 197, 199–200
 - post_pack() 200
 - pre_pack() 200
- packed 196
- packing and unpacking 196–200
 - pack() 197
 - property attributes 196
 - unpack() 198
- PHOLD 67
- plus arguments
 - +vera_continue_on_error 345
 - +vera_debug_on_error 345, 353
 - +vera_exit_on_error 345
 - +vera_finish_on_end 345
 - +vera_load 344
 - +vera_mailbox_size 346
 - +vera_mload 344
 - +vera_region_size 346
 - +vera_semaphore_size 346
 - +vera_stop_on_error 344
 - +vera_text_debug 345
- polymorphism 166
- port
 - See virtual ports
- port variables 73–74
- posedge 81
- postmatch() 55
- PR 67
- prematch() 54
- printf() 141
- prodget() 213
- prodset() 213
- production definition 207–208
- production items 208
- production weights 209
- project files 254, 255
- property attributes 196
- protected class members 162
- PSAMPLE 67
- putc() 52
- Q**
- query() 225, 237–238
- query_str() 238
- R**
- rand 179
- rand_mode() 186
- rand48() 146
- randc 179
- randcase 107
- random numbers 145–146
 - setting seeds at compile time 324

random packet generation 178–190
 random variables 178
 active and inactive random variables 186
 random() 145
 randomize() 180–182
 post_randomize() 181
 pre_randomize() 181
 randseq 207
 reg 46
 region_enter() 125
 region_exit() 126
 regions 125–127
 alloc() 125
 conceptual overview 125
 region_enter() 125
 region_exit() 126
 setting maximum number 323, 346
 timeout() 130
 repeat 107–108
 usage in VSG 211
 return 93
 rewind() 144

S

sample 84
 implicit synchronization 89
 search() 54
 semaphore_get() 122
 semaphore_put() 123
 semaphores 122–125
 alloc() 122
 conceptual overview 122
 semaphore_get() 122
 semaphore_put() 123
 setting maximum number 323, 346
 timeout() 130
 shadow variables 116
 signal declaration 67–79
 HDL nodes 68–69
 interface specification 67–68
 signal depth 69
 signal direction 67
 signal type 67
 signal width 67
 skew 67
 virtual ports 70

signal depth 69
 signal direction 67
 signal types 67
 signal width 67
 signal_connect() 77
 simulation control 149–150
 simulation errors 146–149
 debug support routines 148–149
 error handling 146–148
 skew 67
 sprintf() 58
 sscanf() 58
 state
 declaration
 m_state 227
 state bin name 226
 state declaration 225–228
 all 226
 illegal state declarations 228
 bad_state 228
 m_bad_state 228
 multiple state bin declarations 227
 state bin name 226
 state specification 225
 usage of conditionals 227
 state specification 225
 statement blocks 42
 static variables 94
 stimulus generation
 constraint blocks 186–190
 overview 177–178
 packing and unpacking 196–200
 random packet generation 178–190
 random variables 178
 randomize() 180–182
 stop() 149
 string formats 50
 strings 44, 46, 50–59
 class methods 51–59
 pattern matching 53–56
 type conversion 56–58
 format specifiers 50
 printing strings 50
 string operators 51
 subclasses 159
 subroutines 91–97
 arguments 94–96

- breakpoint 94
- default arguments 95
- external default arguments 97
- external subroutines 97
- optional arguments 96
- return 93
- substr() 53
- superclass 161
- support and feedback 39
- suspend_thread() 115
- sync
 - See synchronization.
- sync() 118–119
- synchronization 81
 - implicit synchronization 89
- T**
- tasks 92–97
 - calling from an HDL 277
 - default arguments 95
 - exporting to HDL 277
 - external default arguments 97
 - external tasks 97
 - optional arguments 96
- template generator
 - for Verilog 309–312
 - for VHDL 319–320
- terminate 114
- text macros 337
- this 157
- thismatch() 55
- timeout() 130
- timescale 255
- trace() 148
- trans 228
- transition declarations 228–232
 - all) 230
 - illegal transition declarations
 - m_bad_trans 231
 - not trans 231
 - illegals transition declarations 231
 - m_trans 230
 - multiple transition bin declarations 230
 - state transition 229
 - transition bin names 229
 - use of conditionals 230
- triggers 117–118
- HAND_SHAKE 118
- ONE_BLAST 118
- ONE_SHOT 117
- typedef 168
- U**
- UDF
 - C/C++ arguments 280
 - declaring in Vera 279
 - handling special events 285
 - linking UDFs to Vera 281–284
 - object files and vera_local.dl 282
 - PLI support procedures 284–285
 - Vera arguments 279
 - Vera UDF table 281
- unpack() 198, 199–200
- unpacked 196
- unpacking
 - See packing and unpacking.
- urand48() 146
- urandom() 145
- V**
- value change alert 87–88
- var
 - usage in UDFs 280
 - usage with HDL tasks 276
- var arguments 95
- variables
 - assignment 63
 - declaration 45–50
 - initialization 45–48
 - shadow variables 116
 - static variables 94
- VCA
 - See value change alert.
- vca() 88
- vcon files
 - See configuration files
- Vera license queueing 34
- Vera Stream Generator
 - See VSG.
- vera.ini 320–325
 - map 324
 - vera_continue_on_error 322
 - vera_debug_on_error 323
 - vera_exit_on_error 322

(427)

- vera_finish_on_end 322
- vera_load 321
- vera_mailbox_size 323
- vera_mload 321
- vera_mload_define 321
- vera_rand48_seed 324
- vera_random_seed 324
- vera_region_size 323
- vera_semaphore_size 323
- vera_stop_on_error 322
- vera_text_debug 323
- vera_free_arg() 285
- vera_get_arg() 284
- vera_local.dl 282
- vera_mti.dl 283
- vera_num_arg() 284
- vera_put_arg() 285
- VERA_REASON 286
- vera_return_value() 285
- Vera-CORE
 - overview 265–266
 - usage model for IP users 267
 - usage model for IP vendors 266
- Vera-CS
 - recompiling Vera-CS 38
- Vera-SV
 - backward compatibility 308
 - C/C++ aspects 303–304
 - initializing connections 295–297
 - managing errors 298
 - master-slave configuration 299
 - overview 293–295
 - servicing remote calls 297
 - submitting remote calls 297–298
 - troubleshooting 307
- Verilog simulators
 - recompiling ModelTech 37–38
 - recompiling Verilog-VCS 34–35
 - recompiling Verilog-XL 35–37
- verilog_node
 - See HDL nodes.
- verilog_task 275
- Verilog-VCS
 - recompiling 34–35
- Verilog-XL
 - recompiling 35–37
- veritask 257
- vhdl_node
 - See HDL nodes
- vhdl_task 275
- virtual classes 164
- virtual ports 70
- void binds 75
- void constraints 189
- void drives 83
- VSG
 - if-else usage 210
 - overview 207
 - prodget() 213
 - prodset() 213
 - production definition 207–208
 - production items 208
 - production weights 209
 - randseq 207
 - usa of break 211
 - usage of case 210
 - usage of continue 212
 - usage of repeat 211
 - value passing 212–215
- VSIM commands 325
- vsv_call_func() 297
- vsv_call_task() 297
- vsv_close_conn() 296
- vsv_get_conn_err() 298
- vsv_make_client() 295
- vsv_make_server() 295
- vsv_up_connections 296
- vsv_wait_for_done() 299
- vsv_wait_for_input() 299
- W
 - wait_child() 113
 - wait_var() 114
 - watch windows 364
 - while 109
 - white space 41

Appendix 3

5

SMART SIMULATION USING COLLABORATIVE FORMAL AND SIMULATION ENGINES

Pei-Hsin Ho
Thomas Shiple
Kevin Harer
James Kukula
Robert Damiano
Valeria Bertacco
Jerry Taylor
Jiang Long

Advanced Technology Group
Synopsys, Inc.

10
15

Smart Simulation

Using Collaborative Formal and Simulation Engines

Pei-Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula,
Robert Damiano, Valeria Bertacco, Jerry Taylor, Jiang Long

{pho, shiple, kevinh, kukula, robertd, valeria, jerryt, long}@synopsys.com

Advanced Technology Group, Synopsys Inc.

Abstract

We present *Ketchum*, a tool that was developed to improve the productivity of simulation-based functional verification by providing two capabilities: (1) *automatic test generation* and (2) *unreachability analysis*. Given a set of "interesting" signals in the design under test (DUT), automatic test generation creates input stimuli that drive the DUT through as many different combinations (called *coverage states*) of these signals as possible to thoroughly exercise the DUT. Unreachability analysis identifies as many unreachable coverage states as possible.

Ketchum differs from the previous published results for several reasons. First, Ketchum provides 10x higher capacity than previous published results. The higher capacity is achieved by carefully orchestrating simulation and multiple formal methods including *symbolic simulation*, *SAT-based BMC*, *symbolic fixpoint computation* and *automatic abstraction*. Second, Ketchum performs not only automatic test generation but also unreachability analysis, which enables the test generation effort to be focused on coverage states that are not unreachable. Third, the backbone of Ketchum is an off-the-shelf commercial simulator. It enables Ketchum to reach deep states of the design quickly and supports simulation monitors through the standard API of the simulator during test generation.

We applied Ketchum to several industrial designs, including the picoJava microprocessor from SUN and the DW8051 microcontroller from Synopsys and obtained very promising results. The experiments show that Ketchum can (1) handle design blocks containing more than 4500 latches and 170K gates, (2) reach up to 6x more coverage states than random simulation and (3) identify a majority of the unreachable coverage states.

1. Introduction

Functional verification checks if the functionality of the hardware design meets the specification. A typical method for "bullet-proofing" the functionality of the design is random simulation [11]. Random simulation can leverage today's fast simulators and computer farms by using some sophisticated test harness. Verification engineers need to build a model of the environment in which the DUT operates. During random simulation, biased pseudo random generators drive the primary inputs of the environment model with random values and the environment model then drives the primary inputs of DUT with random but legal stimuli. The behavior of the DUT can be checked by simulation monitors during the simulation. We call the model that consists of the DUT and the environment model the *model under test (MUT)*.

To measure the quality of the verification effort, verification engineers apply *coverage metrics* to estimate how thoroughly the input stimuli have exercised the design. Coverage metrics directly based on the source code of the RTL design, such as line coverage, are too weak because they do not take the concurrency of hardware designs into consideration. Consider two finite state machines (FSMs) that control a buffer (see Figure 1).

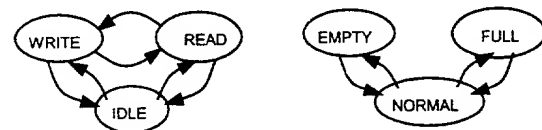


Figure 1. Two FSMs that control a buffer

The first FSM represents the operation that is being performed on the buffer, the second FSM the status of the buffer. It's possible to achieve 100% line coverage on the corresponding RTL description by

exercising only 3 cross-states: idle/empty, write/normal, read/full. However, interesting corner cases such as read/empty (reading when the buffer is empty) and write/full (writing when the buffer is full) could be left unvisited.

State coverage, on the other hand, can distinguish different concurrent events. Given a set of interesting signals, which we call *coverage signals*, the state coverage measures how many different combinations of the values of the coverage signals have been reached during simulation. We call each combination a *coverage state*. For the above example, if we select the set of signals that encodes the states of the two FSMs as our coverage signals, 100% state coverage indicates that all nine cross states have been visited.

Coverage signals are usually selected from signals that constitute pipeline control, interacting FSMs, decoder outputs, status registers or other key control signals. By forcing the design into many different coverage states, the probability of detecting errors is increased. For the above example, a simulation monitor may detect that some valid data in the buffer is mistakenly over-written during a write to a full buffer. Therefore, verification engineers may want to construct input stimuli to maximize the state coverage; that is, to reach as many coverage states as possible. This task is known as *coverage-driven test generation*. Today it can be achieved by manually writing test sequences, or tweaking the biasing in random simulation. However, this approach is very expensive in terms of engineering resources.

Alternatively, formal methods like symbolic fixpoint computation and symbolic simulation can exhaustively search for an input sequence to hit a coverage state. Recently many formal and semi-formal (mixture of formal and simulation techniques) methods [1][7] have been proposed for this application. However, existing results in the literature suffer from a serious capacity limit, as they were only demonstrated on design blocks with less than 500 latches.

Our contribution is the development of a practical solution, Ketchum, to automate the test generation process for verifying real-world designs. Given (1) a synthesizable MUT and (2) a set of less than 64 coverage signals, Ketchum automatically (1) generates test sequences to reach as many coverage states as possible and (2) identifies as many unreachable coverage states as possible. The generated high-coverage tests can be replayed during regression tests. Simulation monitors written in arbitrary languages communicating to the standard API of the commercial simulator can be used during test generation for catching bugs.

The target capacity of Ketchum is to handle synthesizable MUT in the range of 100K gates and 5K latches. This range encompasses design blocks of integral functionality and many IP blocks, to which random simulation may be applied today.

Ketchum's unreachability analysis employs *symbolic fixpoint computation* [12] and robust *automatic abstraction techniques* to prove many coverage states unreachable. Its automatic test generation utilizes a combination of simulation and formal methods like *symbolic simulation* [1][4] and *SAT-based bounded model checking* (SAT-based BMC)[2], to generate input sequences to achieve high state coverage.

More specifically, random simulation and formal methods are interleaved to perform a deep and wide exploration of the state space. Starting from a given initial state, random simulation quickly hits several new coverage states but will eventually stop reaching new coverage states so quickly. Then one of the formal methods is used in an exhaustive search for a new coverage state. This search is narrowed to just those states not yet proven unreachable. The process then repeats, leveraging the ability of simulation to search deep, and the ability of formal methods to search wide.

We applied Ketchum to several industrial designs, including the picoJava microprocessor from SUN and the DW8051 microcontroller from Synopsys and obtained very promising results. The experiments show that Ketchum can (1) handle design blocks containing more than 4500 latches and 170K gates, (2) reach up to 6x more coverage states than random simulation and (3) identify a majority of the unreachable coverage states.

The remainder of the paper proceeds as follows. Section 2 describes our method for generating input sequences to reach coverage states, and Section 3 our algorithm for proving coverage states unreachable. Section 4 presents the experimental results. Having defined the problem and explained our approach in detail, Section 5 describes related work, and finally Section 6 concludes the paper.

2. Test Generation

We present a test generation algorithm that combines (1) random simulation, (2) symbolic simulation and (3) SAT-based bounded model checking to generate stimuli to reach coverage states. We will first provide a brief overview of these three search techniques.

A *state* (resp. *coverage state*) of a design is a valuation of all signals (resp. *coverage signals*) of the design. Given a starting state, random simulation

automatically generates a trace of the MUT from the starting state by driving the primary inputs of the MUT with random values. During the random simulation, we observe the values of the coverage signals. If the values of the coverage signals indicate that a new coverage state has been reached, we store the new coverage state and mark it as reached. The advantage of random simulation is that it is extremely fast and it can generate traces that reach very deep states in the state space. The disadvantage is that it only searches along a single trace at a time. We perform random simulation using a commercial simulator. Note that off-the-shelf simulators usually perform much better in generating a single very long trace than a lot of short traces where the simulator needs to be injected with the same starting state over and over again. We use random simulation as our long range search engine to reach deep states.

Symbolic simulation drives each primary input of the MUT with a new symbolic variable at each simulation step [4]. It computes the symbolic formula for each signal according to the logic in its fanin. Notice that the DUT will be driven by legal symbolic formulas because of the environment model in the MUT. The symbolic formulas are stored as Binary Decision Diagrams (BDDs) [3]. Given a starting state, the i -th step of symbolic simulation can reach any new coverage state that is i steps away from the starting state. We store the unclassified coverage states as a BDD, called *unclassified* BDD. During symbolic simulation, we check if a new coverage state has been reached by substituting the coverage signals in the unclassified BDD with the symbolic formulas of the coverage signals. If the result of the operation is not the empty set, a new coverage state has been reached by symbolic simulation. In that case, we update the unclassified BDD and generate a trace to be used in simulation, as discussed later. Otherwise we continue performing additional steps of symbolic simulation until a new coverage state is reached or some memory or time limit is reached.

Table 1. Comparison of Search Engines

Engine	Effective Search Range	Strength	Limitation
Random simulation	Long	Deep states	Single trace
Symbolic simulation	Medium	Designs with fewer inputs	Time, memory, length of trace
SAT-based BMC	Short	Short hit traces	Time, length of trace

Two important observations about symbolic simulation can be noted. First, the number of symbolic variables that have been used during simulation has more impact on the complexity of symbolic simulation than the number of latches in the

MUT. The number of symbolic variables used in the symbolic simulation is the number of primary inputs times the number of simulation steps. Second, when the size of the BDD that represents the simulation values gets unwieldy, we can easily underapproximate the symbolic simulation values by setting some symbolic variables to Boolean constants [1][6]. As a result of these two observations, we can efficiently perform symbolic simulation on designs with thousands of latches and hundreds of primary inputs for 10 to 50 steps. Because of our first observation, symbolic simulation generally is not adequate to generate traces to reach coverage states that are more than a few tens of steps away from the initial state. Thus, we use symbolic simulation for middle-range exhaustive (or semi-exhaustive with underapproximation) search.

SAT-based BMC was introduced in [2]. Given a particular starting state, we can use a SAT solver to search for a trace up to a certain length i to reach a new coverage state. The targeted coverage states can be restricted to coverage states that are not yet reached or proven unreachable by the unreachability engine. Within a certain time limit, a *complete* SAT solver [14] will: (1) find a trace to reach a new coverage state, or (2) prove that no new coverage states can be reached by traces up to length i , or (3) return no conclusions. If the outcome is (2) or (3), we apply the SAT solver to search for traces of length $i+1$, until some memory or time limit is reached. For designs with thousands of latches and hundreds of inputs, SAT-based BMC usually requires a lot less memory and is a lot faster than symbolic simulation for finding traces of length less than 10. Consequently, we use SAT-based BMC as our short-range exhaustive search engine.



Figure 2. Test generation algorithm

Knowing the advantages and disadvantages of each search engine (summarized in Table 1), we orchestrate the search engines to hide the disadvantages and exploit the advantages of individual engines. We want to perform a deep and wide search by randomly simulating to a deep state

and, starting from that state, a short or middle range wide (exhaustive) search using SAT-BMC or symbolic simulation.

Figure 2 illustrates the test generation algorithm. The rectangular box represents the entire state space of the MUT and the stars represent the coverage states in the state space. The algorithm starts off by playing an initialization sequence provided by the user to reach the initial state of the MUT (a). Then, we perform random simulation (the generated trace is represented as the zig-zag line) and quickly reach easy-to-reach coverage states.

After most of the easy-to-reach coverage states have been reached, the rate of reaching new coverage states falls below a heuristic threshold (b). At this moment, we kick off our SAT-based BMC with the current state of random simulation as the starting state. If SAT-based BMC does not reach a coverage state within the search range, we kick off symbolic simulation after that. The search space of an exhaustive search engine is represented as nested circles in Figure 2. When, the exhaustive search engine finds a new coverage state, it generates a sequence of input assignments that is then replayed by the simulator (represented by the gray arrows). If both exhaustive search engines run out of time or memory limits without finding a new coverage state, we will return control to the random simulator. In either case, we start random simulation again and repeat this process until a desirable coverage number is reached. The final output of Ketchum will be a single long trace that traverses all the reached states, which is stored in a separate file for use during regression testing of the MUT. Since all the traces returned by the formal engines are replayed through the simulator, the simulation monitors will function correctly just as in ordinary random simulation.

We noticed that after an exhaustive engine reaches a new coverage state, the subsequent random simulation run could often reach a bunch of new coverage states quickly. We suspect that the reason is that among the set of coverage signals, some coverage signals are relatively easy to transition from one value to another and some coverage signals are relatively hard to transition. As a result, after an exhaustive engine manages to reach a new combination of the hard-to-transition coverage signals, random simulation will bump into different combinations of the easy-to-transition coverage signals, which combined with the new combination of the hard-to-reach signals become new coverage states.

3. Unreachability

Proving coverage states unreachable is a major aspect of coverage-driven test generation. Knowing the number of unreachable coverage states provides a more accurate measure of the quality of a simulation test bench. Moreover, knowing specifically which states are unreachable allows the reachability engine of Ketchum to focus on a much smaller set of states.

The goal of the unreachability engine of Ketchum is to provide fast and robust results without necessarily trying to detect all of the unreachable states. Simply put, we sacrifice exactness in favor of capacity and robustness. To this end, we have designed a straightforward algorithm that is conservative: it can prove states unreachable, but cannot prove states reachable. The basic idea of the algorithm is one that has been employed by many others before: perform exact analysis on a pruned model of the MUT. However, we introduce novel techniques in the way we select the latches and the combinational logic to include in this pruned model.

Specifically, we first select a small set of latches that includes the coverage signals; the remaining latches are treated as primary inputs. Next, a novel cutting procedure is used to reduce the fanin of the chosen latches, to further simplify the analysis. Then, exact reachability analysis is performed on the pruned model using BDDs. The computed set of reachable states is projected onto the coverage signals. Coverage states in the complement of this projection are provably unreachable, since the pruned model is an abstraction of the original. In the following we detail the latch selection and logic cutting procedures of the algorithm.

The latch selection process starts with the coverage signals. We then add latches incrementally in a breadth-first search (BFS) of the latch dependency graph of the MUT until we reach a user-specified limit on the number of latches. If all the latches of the last BFS level visited cannot be included within the limit, then an arbitrary subset from that level is used.

After a subset of latches has been selected, we employ a cutting algorithm to reduce the number of variables in the support of the transition functions. Experience has shown that even if only 50 or so latches are being analyzed, if the transitive fanin of the latch subset depends on many hundreds of primary inputs and non-selected latches, then later BDD computations will be intractable.

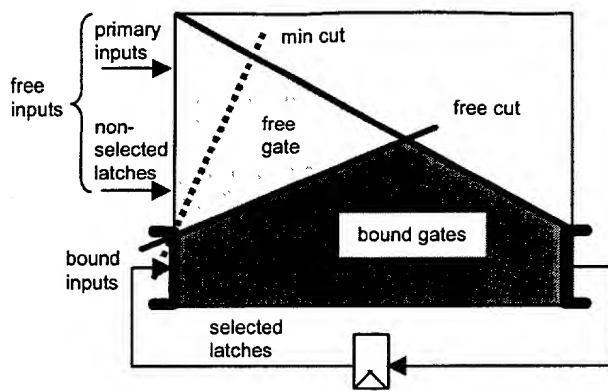


Figure 3. Definition of min-cut for cutting procedure

To explain the algorithm we define a few terms. Consider the transitive fanin of the selected latches (see Figure 3). The *bound inputs* are the outputs of the selected latches. The *free inputs* are the primary inputs and the outputs of the non-selected latches. A gate in the transitive fanin of the selected latches is *bound* if its transitive fanin contains a bound input, otherwise it is *free*. Now consider the free gates that directly feed bound gates; we call the collection of such signals the *free cut*. The signals on the free cut may be correlated, but they do not depend on the bound inputs. Hence, it would appear to be a good tradeoff to replace all the signals on the free cut with primary inputs, thus removing all the free gates from the BDD computations. But, we can do much better, for it is not the number of gates that we want to reduce, but the number of signals in the support of the transition functions.

To this end, we extract a directed network from the signal dependency graph (the graph whose set of nodes is the set of signals and set of directed edges is the fanin relation). The sources of this network are the free inputs, and the sinks are the signals on the free cut. We then compute a minimum cut of this network and replace the signals on the min-cut with primary inputs. The min-cut has the advantage of not only reducing the number of variables in the BDD computations, but also of maintaining more correlation relative to the free cut.

Experience shows that using the min-cut almost always produces the same set of unreachable coverage states as compared to no cutting at all. Furthermore, there are cases where the reachability fixed point computation only completes when using the min-cut. For instance, in the case of IU, one of the experimental results reported below, the cache miss signal originally had 1102 variables in its support, and reachability could not complete. The min-cut procedure reduces the support to 60 variables, and enables reachability to complete with a

result that is superior to that of just leaving the cache miss signal out of the analysis entirely.

It is clear that the unreachable algorithm computes a lower bound on the number of unreachable coverage states of a MUT. In other words, every coverage state that was identified as unreachable is really unreachable, but an unreachable coverage state may be missed due to automatic abstraction.

4. Experimental Results

We implemented a prototype of Ketchum in C. We used a commercial Verilog simulator (VCS) as our random simulator and we developed in house all the other engines. Our symbolic simulator and unreachable engines rely on the CUDD package [18] for BDD computations. The SAT engine is an implementation of the GRASP algorithm [14].

We applied our prototype of Ketchum to some real-world designs. Table 2 summarizes the experimental results on the Integer Unit, Data Cache Unit and Stack Management Unit of the Sun picoJava microprocessor [16], the entire Synopsys DesignWare DW8051 microcontroller, and a commercial bus controller design "Bus". Each experiment was run on a Sun Solaris server of 4 processors and 4GB memory.

The first 3 columns show the characteristics of the five designs. The 2nd column shows the number of latches in each design. The 3rd column shows the number of coverage signals. Note that the number of coverage states is 2 to the number of coverage signals. In order to choose the coverage signals, we located the control FSMs in the source code of the designs. Then we chose the latches that encode these control FSMs as the coverage signals. For example, for the case of DCU, we identified the "cache-miss" FSM (6 latches), the "cache-fill" FSM (5 latches), the "write-back" FSM (8 latches) and the "zero-out" FSM (6 latches). Because all these FSMs are 1-hot encoded, there are at most $6 \times 5 \times 8 \times 6 = 1440$ reachable coverage states. Among those coverage states, Ketchum identified that only 111 are potentially reachable and it was actually able to generate tests that reach all of the 111 coverage states. Note that Ketchum was not told in advance about the FSM encoding.

The 4th column shows the unreachable results: the number of coverage states that were NOT identified by Ketchum as unreachable (the number of potentially reachable coverage states) and the CPU time taken by the unreachable analysis. We can see that Ketchum can effectively identify the majority of the coverage states as unreachable. The number of

latches that we include in the abstract model is 50 across all designs.

In the next 3 columns we compare two automatic test generation approaches, random simulation and Ketchum, in terms of the effectiveness of reaching coverage states. The 5th column shows the number of coverage states reached by random simulation and the time taken by random simulation (24 hours). The 6th column shows the corresponding results for Ketchum. The 7th column shows the percentage improvement in terms of the number of coverage states that Ketchum reached over random simulation. We can see that Ketchum uniformly outperforms random simulation by a large margin. The exception is the DCU unit of picoJava, on which random simulation reached almost as many coverage states as Ketchum but cost 700x more CPU time.

Table 2. Experimental Results

DUT	Ltch	Cov sig.	Cov state aftr Kchm unreach	Reach cover states Rndm	Reach cover states Kchm	Imp (%)
IU	4558	17	230 445sec	9 24hr	40 24hr	344
8051	784	11	896 299sec	317 24hr	597 24hr	88
DCU	385	25	111 259sec	109 24hr	111 2min	2
SMU	217	16	132 1423sec	104 24hr	132 45min	30
Bus	155	16	342 60sec	44 24hr	342 75min	677

During the experiments, we provided minimum environment models for the designs. We set some exception signals (like the test signals that control the scan chain) to constants and assigned very small probabilities for asserting some reset signals during random simulation. The same setting and biasing were applied to both random simulation and Ketchum. So we believe that it is a fair comparison.

Since Ketchum needs to analyze the MUT that consists of both the DUT and the environment model for test generation, the size of the DUT that Ketchum can effectively handle decreases as the size of the environment model increases. In our experience, since the environment model only needs to model the interface behavior of the DUT, a reasonable environment model of the DUT is usually much simpler than the DUT itself, especially if the DUT is an integral functional block of the design.

5. Related Work

Unreachability Analysis

Symbolic fixpoint computation [12] is the common basis for unreachability analysis. Different methods that over-approximate the reachable state space have been proposed in [1][5][8][13]. However, these methods were designed for approximating reachable "states", not reachable "coverage states", so they do not utilize the fact that the logic that is closer to the coverage signals has more impact on the behavior of the coverage signals than the logic that is farther away. The algorithms presented in [15] perform unreachability analysis on the pruned model where all latches except the coverage signals of the original DUT are treated as primary inputs. However, this may leave too many primary inputs to perform symbolic fixpoint computation. In [9], a heuristic is introduced to reduce the number of primary inputs. Our algorithm, on the other hand, finds the optimal cut to minimize the number of primary inputs.

Test Generation

Algorithms that perform some form of over-approximated symbolic image computation to guide simulation for reaching coverage goals have been proposed in [1][8][17]. However, these solutions have been ineffective so far in attacking designs with thousands of latches, on which even the first image computation may not terminate. In [1] under-approximated symbolic simulation was introduced to mitigate BDD blowup. But according to our experimental results, the under-approximation technique presented in the paper is too drastic in the sense that it often misses coverage states.

Our test generation algorithm is similar to the SIVA system described in [7] in the sense that both methods interleave simulation and formal engines to reach coverage goals. However, the following are the major differences. First, SIVA uses ATPG and symbolic image computation while Ketchum uses symbolic simulation and SAT-based BMC. We believe that using symbolic image computation for test generation is not practical. On the simulation side, SIVA computes a search tree rather than a linear trace, which prevents it from taking advantage of the speed and the deep states offered by commercial simulators. In addition, with the commercial simulator, Ketchum can utilize arbitrary simulation checkers. In terms of coverage goals, SIVA was designed for maximizing toggle coverage rather than state coverage. But in practice, toggle coverage goals in the DUT are often relatively easy to reach, so the edge of SIVA over random simulation is slim. At last, it was reported that SIVA was only tested on examples with around 400 latches whereas we

successfully applied Ketchum on real-world designs with 10x the number of latches.

6. Conclusion and Future Work

We have presented Ketchum, a tool that automates the problem of coverage-driven test generation. This novel technology combines multiple formal verification techniques and random simulation to classify most of the unreachable coverage states and reach up to 6x more coverage states than random simulation alone. At the same time, it can handle designs of more than 4500 latches, an order of magnitude more complex than published formal or semi-formal verification results.

Through the use of a robust abstraction algorithm, Ketchum is able to quickly prove most coverage states as unreachable. Central to this algorithm is a cutting procedure that reduces the variable support of the transition functions of the abstract model, enabling unreachability analysis to complete on big designs. On the reachability side, Ketchum employs a novel interleaving of simulation and formal verification techniques that exploits hard-to-reach coverage states. The combination of these approaches greatly improves the mobility of the search in the state space, thus leading to better coverage results.

There are multiple directions to extend and improve Ketchum. For coverage metrics, we want to extend Ketchum to handle transition coverage. For unreachability, foremost is an enhanced algorithm for selecting a subset of latches that takes more factors into account beyond BFS level, such as the number of fanins from, and fanouts to, the coverage signals. For test generation, we want to improve the underapproximation capabilities of the formal techniques, as well as integrate additional engines, such as sequential ATPG.

7. References

- [1] J. Bergmann and M. Horowitz. Improving coverage analysis and test generation for large designs. In Proceedings of ICCAD, 1999.
- [2] V. Bertacco, M. Damiani and S. Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In Proceedings of DAC, pp. 391-396, 1999.
- [3] A. Biere, A. Cimatti, E. Clarke, M. Fujita and Y. Zhu. Symbolic model checking using SAT procedures. In Proceedings of DAC, 1999.
- [4] R.E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, C-35(8), pp 677-691, 1986.
- [5] H. Cho, G. Hatchel, E. Macii, M. Poncino, and F. Somenzi. Automatic state space decomposition for approximate FSM traversal based on circuit analysis. IEEE TCAD, 15(12), pp. 1451-1464, 1996.
- [6] D.L. Dill. Embedded tutorial: formal verification meets simulation. In Proceedings of ICCAD, 1999.
- [7] M.K. Ganai, A. Aziz and A. Kuehlmann. Enhancing simulation with BDDs and ATPG. In Proceedings of DAC, pp. 385-390, 1999.
- [8] S.G. Govindaraju, D.L. Dill, A.J. Hu, and M.A. Horowitz. Approximate reachability with BDDs using overlapping projections. In Proceedings of DAC, pp. 451-455, 1998.
- [9] P.-H. Ho, A. Isles and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In ICCAD, 1998.
- [10] R.C. Ho and M. Horowitz. Validation coverage analysis for complex digital designs. In ICCAD, 1996.
- [11] M. Kantrowitz and L. Noack. I'm Done Simulating: Now what? verification coverage analysis and correctness checking of the DEC chip 21164 Alpha microprocessor. In DAC, pp. 325-330, 1996.
- [12] K.L. McMillan. Symbolic model checking. Kluwer Academic Publishers, 1994.
- [13] I.-H. Moon, J. Kukula, T. Shiple and F. Somenzi. Least fixpoint approximation for reachability analysis. In Proceedings of ICCAD, pp. 41-44, 1999.
- [14] J.P. Marques-Silva and K.A. Sakallah. GRASP: a search algorithm for propositional satisfiability. In IEEE Transaction on Computers, pp. 506-521, May 1999.
- [15] D. Moundanos, J.A. Abraham and Y.V. Hoskote. Abstraction techniques for validation coverage analysis and test generation. In IEEE Transactions on Computers, January 1998.
- [16] Sun Microsystems. PicoJava technology. <http://www.sun.com/microelectronics/communitysource/picojava>.
- [17] C.H. Yang and D.L. Dill. Validation with guided search of the state space. In DAC, pp. 599-604, 1998.
- [18] F. Somenzi. CUDD: CU Decision Diagram Package. <ftp://vlsi.colorado.edu/pub/>.